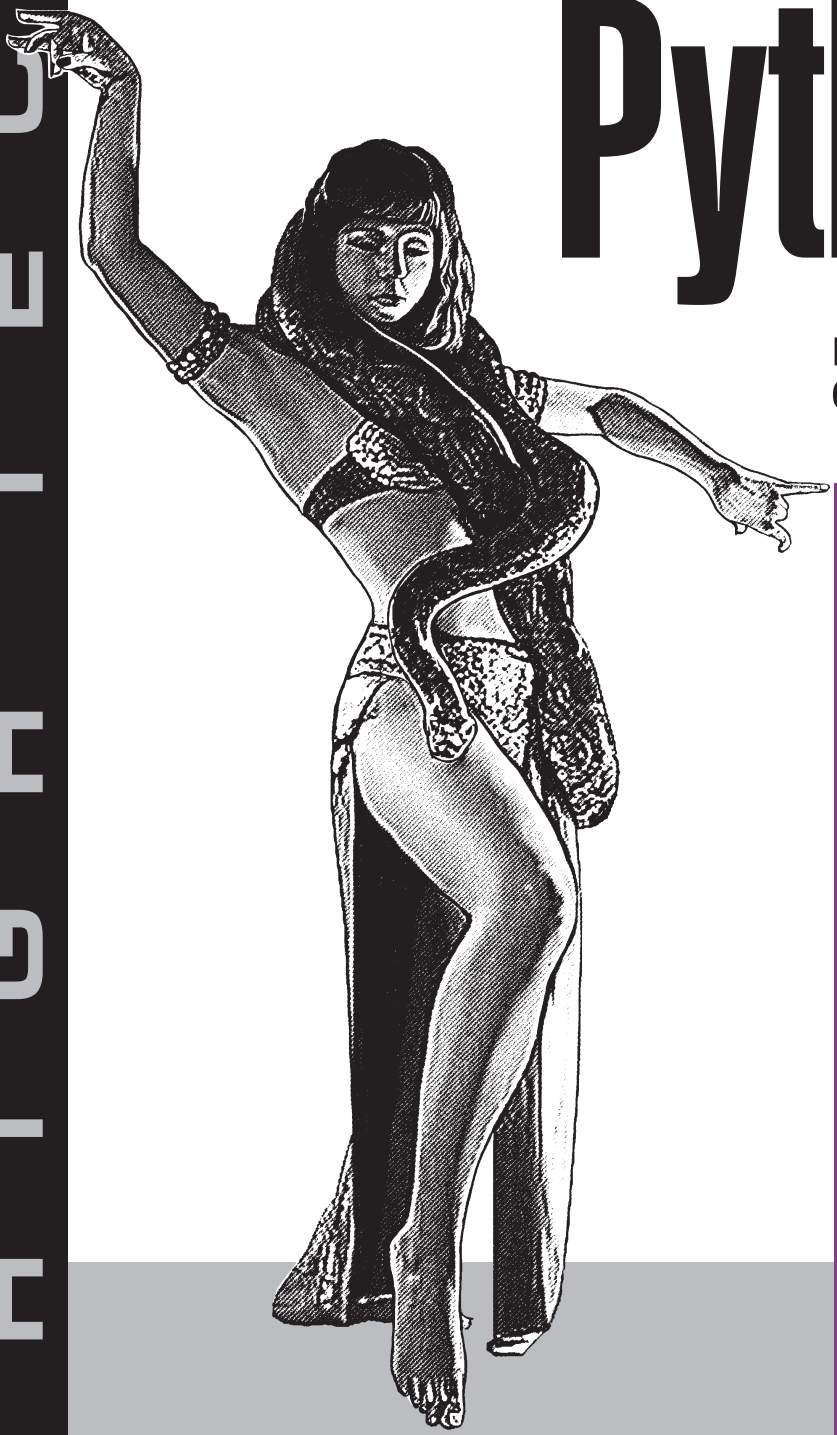
HIGH TECH

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
```

```
        if matchtext in line:
            print line
```

Чтобы воспользоваться описанной функцией, сначала необходимо вызвать ее, затем добиться перемещения потока выполнения до первой инструкции (yield), а потом начать передачу данных с помощью метода send(). Например:

```
>>> matcher = print_matches("python")
>>> matcher.next()     # Перемещение до первой инструкции (yield)
Поиск подстроки python
>>> matcher.send("Hello World")
>>> matcher.send("python is cool")
python is cool
>>> matcher.send("yow!")
>>> matcher.close()    # Завершение работы с объектом matcher
>>>
```

Выполнение сопрограммы приостанавливается до тех пор, пока ей не будет передано новое значение с помощью метода send(). Когда это произойдет, выражение (yield) внутри сопрограммы вернет полученное значение и ее работа продолжится со следующей инструкции. Сопрограмма будет выполняться, пока не будет встречено следующее выражение (yield) — в этой точке выполнение функции будет приостановлено. Этот процесс будет продолжаться, пока либо сопрограмма сама не вернет управление, либо пока не будет вызван метод close(), как было показано в предыдущем примере.

Сопрограммы удобно использовать при разработке многозадачных программ, работа которых укладывается в схему «производитель-потребитель», когда одна часть программы производит некоторые данные, а другая часть потребляет их. В этой схеме сопрограммам отводится роль «потребителя» данных. Ниже приводится пример программы, в которой совместно действуют генератор и сопрограмма:

```
# Множество сопрограмм поиска
matchers = [
    print_matches("python"),
    print_matches("guido"),
    print_matches("jython")
]

# Подготовка всех подпрограмм к последующему вызову метода next()
for m in matchers: m.next()

# Передать активный файл журнала всем сопрограммам.
# Обратите внимание: для нормальной работы необходимо,
# чтобы веб-сервер активно записывал данные в журнал.
wwwlog = tail(open("access-log"))
for line in wwwlog:
    for m in matchers:
        m.send(line) # Передача данных каждой из сопрограмм
```

Дополнительная информация о сопрограммах приводится в главе 6.

Объекты и классы

Все значения, используемые программами, являются объектами. Объект содержит некоторые внутренние данные и обладает методами, позволяющими выполнять различные операции над этими данными. Вам уже приходилось использовать объекты и методы, когда вы работали с данными встроенных типов, такими как строки и списки. Например:

```
items = [37, 42] # Создание списка объектов
items.append(73) # Вызов метода append()
```

Функция `dir()` выводит список всех методов объекта и является удобным инструментом для проведения экспериментов в интерактивной оболочке. Например:

```
>>> items = [37, 42]
>>> dir(items)
['__add__', '__class__', '__contains__', '__delattr__', '__delitem__',
...
 'append', 'count', 'extend', 'index', 'insert', 'pop',
 'remove', 'reverse', 'sort']
>>>
```

При исследовании объектов можно заметить уже знакомые методы, такие как `append()` и `insert()`. Кроме того, в списках можно также увидеть специальные методы, имена которых всегда начинаются и заканчиваются двумя символами подчеркивания. Эти методы реализуют различные операторы языка. Например, метод `__add__()` реализует оператор +:

```
>>> items.__add__([73,101])
[37, 42, 73, 101]
>>>
```

Для определения новых типов объектов и в объектно-ориентированном программировании используется инструкция `class`. Например, ниже приводится определение простого класса стека, обладающего методами `push()`, `pop()` и `length()`:

```
class Stack(object):
    def __init__(self):    # Инициализация стека
        self.stack = [ ]
    def push(self,object):
        self.stack.append(object)
    def pop(self):
        return self.stack.pop()
    def length(self):
        return len(self.stack)
```

В первой строке определения класса инструкция `class Stack(object)` объявляет, что `Stack` – это объект. В круглых скобках указывается, наследником какого класса является объявляемый класс; в данном случае класс `Stack` наследует свойства и методы класса `object`, который является родоначальником всех типов данных в языке Python. Внутри объявления класса определяются методы, с помощью инструкции `def`. Первый аргумент любого ме-

тода всегда ссылается на сам объект. В соответствии с соглашениями этому аргументу обычно дается имя `self`. Все методы, выполняющие операции над атрибутами объекта, должны явно ссылаться на переменную `self`. Методы, имена которых начинаются и заканчиваются двумя символами подчеркивания, являются специальными методами. Например, метод `__init__()` используется для инициализации объекта после его создания.

Ниже приводится пример использования класса:

```
s = Stack()        # Создать стек
s.push("Dave")     # Поместить на стек некоторые данные
s.push(42)
s.push([3,4,5])
x = s.pop()        # переменная x получит значение [3,4,5]
y = s.pop()        # переменная y получит значение 42
del s              # Уничтожить объект s
```

В этом примере был создан совершенно новый объект, реализующий стек. Однако по своей природе стек очень похож на встроенный тип списка. Поэтому можно было бы унаследовать класс `list` и добавить один дополнительный метод:

```
class Stack(list):
    # Добавить метод push() интерфейса стека
    # Обратите внимание: списки уже имеют метод pop().
    def push(self,object):
        self.append(object)
```

Обычно все методы, присутствующие в объявлении класса, могут применяться только к экземплярам данного класса (то есть к созданным объектам). Однако существует возможность определять методы различных видов, например статические, знакомые программистам по языкам C++ и Java:

```
class EventHandler(object):
    @staticmethod
    def dispatcherThread():
        while (1):
            # Ожидание запроса
            ...

EventHandler.dispatcherThread() # Вызов метода как обычной функции
```

В данном случае `@staticmethod` объявляет следующий за ним метод статическим. `@staticmethod` – это пример использования *декоратора*, подробнее о которых будет рассказываться в главе 6.

Исключения

При возникновении ошибки в программе возбуждается исключение и выводится трассировочная информация, такая, как показано ниже:

```
Traceback (most recent call last):
 File "foo.py", line 12, in <module>
IOError: [Errno 2] No such file or directory: 'file.txt'
```

```
(Перевод:
Трассировочная информация (самый последний вызов – самый нижний):
 Файл “foo.py”, строка 12, в <модуль>
IOError: [Errno 2] Нет такого файла или каталога: ‘file.txt’
)
```

Трассировочная информация позволяет определить тип ошибки и место, где она возникла. Обычно ошибки приводят к аварийному завершению программы. Однако имеется возможность перехватить и обработать исключение – с помощью инструкций `try` и `except`, как показано ниже:

```
try:
    f = open(“file.txt”,”r”)
except IOError as e:
    print e
```

Если будет возбуждено исключение `IOError`, информация о причинах появления ошибки будет записана в переменную `e` и управление будет передано первой инструкции в блоке `except`. Если возникнет какое-то другое исключение, оно будет передано объемлющему блоку кода (если таковой имеется). Если операция выполнена без ошибок, инструкции в блоке `except` будут просто игнорироваться. После обработки исключения выполнение программы продолжится с первой инструкции, следующей за последней инструкцией в блоке `except`. Программа не возвращается в то место, где возникло исключение.

Возбудить исключение вручную можно с помощью инструкции `raise`. Для этого инструкции `raise` можно передать один из встроенных классов исключений, например:

```
raise RuntimeError(“Компьютер говорит нет”)
```

Или создать и использовать собственный класс исключения, как описано в разделе «Определение новых исключений» в главе 5 «Структура программы и управление потоком выполнения».

Управление системными ресурсами, такими как блокировки, файлы и сетевые соединения, часто является достаточно сложным делом, особенно когда при этом необходимо предусматривать обработку исключений. Упростить решение таких задач программирования можно с помощью инструкции `with`, применяя ее к объектам определенного типа. Ниже приводится пример программного кода, который использует мьютекс в качестве блокировки:

```
import threading
message_lock = threading.Lock()
...
with message_lock:
    messages.add(newmessage)
```

В этом примере при выполнении инструкции `with` автоматически приобретается объект блокировки `message_lock`. Когда поток выполнения покидает контекст блока `with`, блокировка автоматически освобождается. Благодаря этому обеспечивается надежное управление блокировкой, независимо от того, что происходит внутри блока `with`. Например, даже если при выпол-

нении инструкций внутри блока возникнет исключение, блокировка все равно будет освобождена после выхода за его пределы.

Обычно инструкция `with` совместима только с объектами, представляющими те или иные системные ресурсы или среду выполнения, такие как файлы, соединения и блокировки. Однако имеется возможность реализовать поддержку этой инструкции в своих классах. Подробнее об этом рассказывается в разделе «Протокол управления контекстом» главы 3 «Типы данных и объекты».

Модули

Часто по мере роста программы возникает желание разбить ее на несколько файлов, чтобы упростить ее разработку и дальнейшее сопровождение. Язык Python позволяет помещать определения в файлы и использовать их как модули, которые могут импортироваться другими программами и сценариями. Чтобы создать модуль, необходимо поместить соответствующие инструкции и определения в файл с тем же именем, которое будет присвоено модулю. (Обратите внимание, что имя файла должно иметь расширение `.py`.) Например:

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

Использовать модуль в других программах можно с помощью инструкции `import`:

```
import div
a, b = div.divide(2305, 29)
```

Инструкция `import` создает новое пространство имен и внутри этого пространства имен выполняет все инструкции, находящиеся в файле с расширением `.py`. Чтобы получить доступ к содержимому этого пространства имен после импортирования, достаточно просто использовать имя модуля в качестве префикса, как это сделано в вызове функции `div.divide()` в предыдущем примере.

Если потребуется импортировать модуль под другим именем, достаточно добавить в инструкцию `import` дополнительный квалификатор `as`, как показано ниже:

```
import div as foo
a,b = foo.divide(2305,29)
```

Чтобы импортировать некоторые определения в текущее пространство имен, можно воспользоваться инструкцией `from`:

```
from div import divide
a,b = divide(2305,29)     # Префикс div больше не нужен
```

Чтобы загрузить в текущее пространство имен все содержимое модуля, можно использовать следующий прием:

```
from div import *
```

При вызове функции `dir()` с именем модуля она выведет содержимое указанного модуля, что делает эту фунцию удобным инструментом для экспериментов в интерактивной оболочке:

```
>>> import string
>>> dir(string)
['__builtins__', '__doc__', '__file__', '__name__', '_idmap',
 '_idmapL', '_lower', '_swapcase', '_upper', 'atof', 'atof_error',
 'atoi', 'atoi_error', 'atol', 'atol_error', 'capitalize',
 'capwords', 'center', 'count', 'digits', 'expandtabs', 'find',
...
>>>
```

Получение справки

При работе с интерпретатором Python у вас всегда под рукой будет несколько источников справочной информации. Во-первых, при работе в интерактивном режиме с помощью команды `help()` можно получить информацию о любом встроенном модуле и о других аспектах языка Python. Можно просто ввести команду `help()` и получить общую справочную информацию или команду `help('`*имя_модуля*`')`, чтобы получить информацию о конкретном модуле. Если команде `help()` передать имя определенной функции, она может вернуть информацию об этой функции.

Большинство функций в языке Python снабжены строками документирования, описывающими порядок их использования. Чтобы вывести строку документирования, достаточно просто вывести содержимое атрибута `__doc__`. Например:

```
>>> print issubclass.__doc__
issubclass(C, B) -> bool

Return whether class C is a subclass (i.e., a derived class) of class B.
When using a tuple as the second argument issubclass(X, (A, B, ...)),
is a shortcut for issubclass(X, A) or issubclass(X, B) or ... (etc.).
(Перевод
Проверяет, является ли класс C подклассом (то есть наследует) класса B.
Когда во втором аргументе передается кортеж - issubclass(X, (A, B, ...)),
такой вызов является сокращенным вариантом issubclass(X, A) or issubclass(X, B)
or ... (и т. д.).
)
>>>
```

И последнее, но также важное замечание – большинство инсталляций Python включает также команду `pydoc`, которая может использоваться для получения документации о модулях Python. Для этого достаточно просто ввести команду `pydoc` *`topic`* в системной командной строке.

2

Лексические и синтаксические соглашения

В этой главе рассматриваются синтаксические и лексические соглашения, используемые в программах на языке Python. Здесь мы познакомимся со структурой строки, способами группировки инструкций, зарезервированными словами, литералами, операторами, лексемами и кодировками символов в исходных текстах.

Структура строк и отступы

Каждая инструкция в программе завершается символом перевода строки. Длинные инструкции допускается располагать в нескольких строках, используя символ продолжения строки (\), как показано ниже:

```
a = math.cos(3 * (x - n)) + \
    math.sin(3 * (y - n))
```

Символ продолжения строки не используется внутри строк в тройных кавычках, в определениях списков, кортежей или словарей. В общем случае любая часть программы, заключенная в круглые (...), квадратные [...], фигурные {...} скобки или в тройные кавычки, может занимать несколько строк без применения символа продолжения строки, потому что в данных случаях явно обозначены начало и конец блока определения.

Отступы используются для отделения различных блоков программного кода, таких как тело функции, условного оператора, цикла или определения класса. Величина отступа для первой инструкции в блоке может выбираться произвольно, но отступы всех остальных инструкций в блоке должны быть равны отступу в первой инструкции. Например:

```
if a:
    инструкция1 # Правильное оформление отступов
    инструкция2
```

```
else:
    инструкция3
      инструкция4 # Неправильное оформление отступов (ошибка)
```

Если тело функции, цикла, условного оператора или определения класса достаточно короткое и содержит единственную инструкцию, его можно поместить в той же самой строке, например:

```
if a: инструкция1
else: инструкция2
```

Для обозначения пустого тела или блока используется инструкция `pass`. Например:

```
if a:
    pass
else:
    инструкции
```

Для оформления отступов допускается использовать символы табуляции, однако такая практика не приветствуется. Для этих целей в сообществе Python предпочитают (и рекомендуется) использовать символы пробела. Когда интерпретатор Python встречает символы табуляции, он замещает их символами пробела до ближайшей позиции в строке, с порядковым номером, кратным 8 (например, когда интерпретатор встречает символ табуляции в позиции с номером 11, он замещает его символами пробела до позиции с номером 16). Если интерпретатор Python запустить с ключом `-t`, он будет выводить предупреждающие сообщения, когда ему будут встречаться строки, где отступы в одном и том же блоке кода оформлены неодинаковой смесью из символов пробелов и табуляции. Если использовать ключ `-tt`, вместо предупреждающих сообщений будет возбуждаться исключение `TabError`.

Допускается размещать в одной строке программы сразу несколько инструкций, при этом они должны отделяться друг от друга точкой с запятой (`;`). Строки, содержащие единственную инструкцию, также могут завершаться точкой с запятой, хотя это и не является обязательным требованием.

Символ `#` отмечает начало комментария, простирающегося до конца строки. Однако, если символ `#` встречается внутри строки в кавычках, он не воспринимается как начало комментария.

Наконец, интерпретатор игнорирует все пустые строки, если он запущен не в интерактивном режиме. Ввод пустой строки в интерактивном режиме интерпретируется как окончание ввода многострочной инструкции.

Идентификаторы и зарезервированные слова

Идентификаторы – это имена, используемые для идентификации переменных, функций, классов, модулей и других объектов. Идентификаторы могут включать алфавитно-цифровые символы и символ подчеркивания (_), но при этом они не могут начинаться с цифрового символа. В качестве

алфавитных символов в настоящее время допускается использовать только символы A–Z и a–z из набора ISO-Latin. Идентификаторы чувствительны к регистру символов, поэтому имена `FOO` и `foo` считаются различными. В идентификаторах не допускается использовать специальные символы, такие как `$`, `%` и `@`. Кроме того, такие слова, как `if`, `else` и `for` являются зарезервированными и не могут использоваться в качестве идентификаторов. В следующем списке перечислены все зарезервированные слова:

```
and          del          from         nonlocal     try
as           elif         global       not          while
assert       else         if           or           with
break        except       import       pass         yield
class        exec         in           print
continue     finally      is           raise
def          for          lambda       return
```

Идентификаторы, начинающиеся или оканчивающиеся символом подчеркивания, часто имеют особый смысл. Например, идентификаторы, начинающиеся с одного символа подчеркивания, такие как `_foo`, не импортируются инструкцией `from module import *`. Идентификаторы, начинающиеся и оканчивающиеся двумя символами подчеркивания, такие как `__init__`, зарезервированы для имен специальных методов. Идентификаторы, начинающиеся двумя символами подчеркивания, такие как `__bar`, используются для реализации частных методов и атрибутов классов, о чем подробно рассказывается в главе 7 «Классы и объектно-ориентированное программирование». В общем случае следует избегать использования одинаковых идентификаторов.

Числовые литералы

В языке Python существует четыре встроенных типа числовых литералов:

- Булевы значения
- Целые числа
- Числа с плавающей точкой
- Комплексные числа

Идентификаторы `True` и `False` интерпретируются как логические значения, имеющие числовое выражение 1 и 0 соответственно. Такие числа, как `1234`, интерпретируются как десятичные целые числа. Чтобы определить число в восьмеричной, шестнадцатеричной или двоичной форме, необходимо задать префикс `0`, `0x` или `0b` соответственно (например, `0644`, `0x100fea8` или `0b11101010`).

Целые числа в языке Python могут содержать произвольное количество цифр, поэтому, если потребуется записать очень большое число, достаточно просто указать все цифры, составляющие его, например: `12345678901234567890`.

Однако при инспектировании таких значений[1] и в старом программном коде можно заметить, что большие числа оканчиваются символом `l` (символ `L` в нижнем регистре) или `L`, например: `12345678901234567890L`. Наличие этого символа `L` связано с тем, что интерпретатор Python представляет целые числа как целые значения с фиксированной точностью, ограниченной аппаратными возможностями, или как целые числа с произвольной точностью, в зависимости от величины значения. В ранних версиях Python, чтобы определить большое число, необходимо было явно добавить завершающий символ `L`. В настоящее время делать это не требуется и даже не рекомендуется. Поэтому, при необходимости определить большое числовое значение, достаточно просто записать его (без символа `L`).

Такие числа, как `123.34` и `1.2334e+02`, интерпретируются как числа с плавающей точкой. Целые числа и числа с плавающей точкой, завершающиеся символом `j` или `J`, такие как `12.34J`, рассматриваются как мнимые части комплексных чисел. Комплексные числа, состоящие из действительной и мнимой частей, определяются как сумма действительного и мнимого числа: `1.2 + 12.34J`.

Строковые литералы

Строковые литералы используются для определения последовательностей символов и оформляются как текст, заключенный в апострофы (`'`), кавычки (`"`) или тройные кавычки (`'''` или `"""`). Нет никакой разницы, какие кавычки использовать, главное, чтобы совпадал тип кавычек в начале и в конце строкового литерала. Определения строковых литералов в апострофах и в кавычках должны занимать не более одной строки, тогда как литералы в тройных кавычках могут занимать несколько строк и могут содержать все символы форматирования (то есть символы перевода строки, табуляции, пробелы и так далее). Соседние строковые литералы (которые отделяются друг от друга пробелом, символом перевода строки или символом продолжения строки), такие как `"hello" 'world'`, при выполнении программы объединяются в одну строку, например: `"helloworld"`.

Внутри строковых литералов символ обратного слэша (`\`) используется для экранирования специальных символов, таких как перевод строки, сам символ обратного слэша, кавычек и непечатаемых символов. Все допустимые экранированные последовательности символов перечислены в табл. 2.1. Неопознанные экранированные последовательности остаются в строке в неизменном виде и включают ведущий символ обратного слэша.

Таблица 2.1. Стандартные экранированные последовательности

Символ	Описание
`\`	Символ продолжения строки
`\\`	Символ обратного слэша
`\'`	Апостроф

[1] Например, функцией `repr()`. – *Прим. перев.*

Таблица 2.1 (продолжение)

Символ	Описание
\"	Кавычка
\a	Сигнал
\b	Забой
\e	Экранирующий символ
\0	Пустой символ
\n	Перевод строки
\v	Вертикальная табуляция
\t	Горизонтальная табуляция
\r	Возврат каретки
\f	Перевод формата
ООО	Восьмеричное значение (от \000 до \377)
\u*хххх*	Символ Юникода (от \u0000 до \uffff)
\U*хххххххх*	Символ Юникода (от \U00000000 до \Uffffffff)
\N{*имя символа*}	Символ Юникода с указанным именем
\xhh	Шестнадцатеричное значение (от \x00 до \xff)

Экранированные последовательности \000 и \x используются для вставки в строковые литералы символов, которые сложно ввести с клавиатуры (то есть управляющие символы, непечатаемые символы, псевдографические символы, символы из национальных алфавитов и так далее). В этих экранированных последовательностях должны указываться целочисленные значения, соответствующие кодам символов. Например, если потребуется определить строковый литерал со словом «Jalapeño», его можно записать, как "Jalape\xf1o", где \xf1 – это код символа ñ.

В версии Python 2 строковые литералы могут содержать только 8-битные символы. Серьезным ограничением таких строковых литералов является отсутствие полноценной поддержки национальных символов и Юникода. Для преодоления этого ограничения в Python 2 используется отдельный строковый тип. Чтобы записать строковый литерал, содержащий символы Юникода, необходимо добавить префикс u перед открывающей кавычкой. Например:

```
s = u"Jalape\u00f1o"
```

В Python 3 этот префикс можно не указывать (в действительности его наличие приводит к синтаксической ошибке), поскольку все строки уже интерпретируются как строки Юникода. В версии Python 2 это поведение можно имитировать, если вызвать интерпретатор с ключом -U (в этом случае все строковые литералы будут интерпретироваться как строки Юникода, и префикс u можно опустить).

Независимо от используемой версии Python экранированные последовательности \u, \U и \N из табл. 2.1 могут использоваться для вставки произвольных символов в строковые литералы Юникода. Каждому символу Юникода присвоен свой *кодовый пункт*, который обычно может быть записан как U+*XXXX*, где *XXXX* – последовательность из четырех или более шестнадцатеричных цифр. (Обратите внимание: такая форма записи не имеет отношения к синтаксису языка Python, но часто используется для описания символов Юникода.) Например, символ с имеет кодовый пункт U+00F1. Экранированная последовательность \u используется для вставки символов Юникода с кодовыми пунктами в диапазоне от U+0000 до U+FFFF (например, \u00f1). Экранированная последовательность \U используется для вставки символов Юникода с кодовыми пунктами от U+10000 и выше (например, \U00012345). Относительно экранированной последовательности \U важно отметить, что символы Юникода с кодовыми пунктами выше U+10000 обычно могут быть представлены в виде пары символов, которая называется *суррогатной парой*. Эта тема касается внутреннего представления строк Юникода и подробно рассматривается в главе 3 «Типы данных и объекты».

Кроме того, символы Юникода имеют описательные имена. Если имя символа известно, можно использовать экранированную последовательность \N{*имя символа*}. Например:

```
s = u"Jalape\N{LATIN SMALL LETTER N WITH TILDE}o"
```

Исчерпывающий справочник по кодовым пунктам и именам символов можно найти по адресу *http://www.unicode.org/charts*.

Дополнительно строковые литералы могут предваряться префиксом r или R, например: r'\d'. Такие строки называют «сырыми», так как в них символы обратного слэша не имеют специального значения, то есть в таких строках никакие символы не интерпретируются особым образом, включая и символы обратного слэша. Основное назначение «сырых» строк состоит в том, чтобы обеспечить возможность определения строковых литералов, где символы обратного слэша могут иметь какой-то другой смысл. В качестве примеров можно привести регулярные выражения, используемые модулем re, или имена файлов в операционной системе Windows (например, r'c:\newdata\tests').

«Сырые» строки не могут состоять из единственного символа обратного слэша, r"\". Внутри «сырых» строк экранированные последовательности вида \u*XXXX* по-прежнему интерпретируются как символы Юникода при условии, что последовательность, следующая за символом \, может интерпретироваться как кодовый пункт. Например, литерал ur"\u1234" определяет «сырую» строку Юникода с единственным символом U+1234, тогда как литерал ur"\\u1234" определяет строку из семи символов, где первые два символа – это символы обратного слэша, а остальные пять – литерал "u1234". Кроме того, в определениях «сырых» строк Юникода в версии Python 2.2 префикс r должен следовать за префиксом u, как показано в примерах. В Python 3.0 префикс u указывать не требуется.

Строковые литералы не должны содержать определения символов в виде последовательностей байтов, соответствующих символам в определенной

кодировке, такой как UTF-8 или UTF-16. Например, при попытке вывести «сырую» строку в кодировке UTF-8, например 'Jalape\xc3\xb1o', будет выведена строка из девяти символов U+004A, U+0061, U+006C, U+0061, U+0070, U+0065, U+00C3, U+00B1, U+006F, что, вероятно, не совсем соответствует ожиданиям. Дело в том, что в кодировке UTF-8 двухбайтовая последовательность \xc3\xb1 должна представлять символ U+00F1, а не два символа U+00C3 и U+00B1. Чтобы определить литерал в виде кодированной последовательности байтов, необходимо добавить префикс "b", например: b"Jalape\xc3\xb1o". В этом случае создается строковый литерал, состоящий из последовательности отдельных байтов. Эту последовательность можно преобразовать в обычную строку, декодировав значение литерала с байтовыми данными с помощью метода decode(). Подробнее об этом рассказывается в главе 3 и в главе 4 «Операторы и выражения».

Надобность в байтовых литералах возникает крайне редко, поэтому данный синтаксис отсутствует в версиях ниже Python 2.6, но и в этой версии не делается различий между байтовыми литералами и строками. Однако в Python 3 байтовые литералы отображаются в новый тип данных bytes, который по своему поведению отличается от обычных строк (см. приложение А, «Python 3»).

Контейнеры

Значения, заключенные в квадратные [...], круглые (...) или фигурные {...} скобки, обозначают коллекции объектов, представленные в виде списка, кортежа или словаря соответственно, как показано в следующем примере:

```
a = [ 1, 3.4, 'hello' ] # Список
b = ( 10, 20, 30 )      # Кортеж
c = { 'a': 3, 'b': 42 } # Словарь
```

Литералы списков, кортежей и словарей могут располагаться в нескольких строках программы, без использования символа продолжения строки (\). Кроме того, допускается оставлять завершающую запятую после последнего элемента. Например:

```
a = [ 1,
      3.4,
      'hello',
    ]
```

Операторы, разделители и специальные символы

В языке Python имеются следующие операторы:

```
+       -       *       **      /       //      %       <<      >>      &       |
^       ~       <       >       <=      >=      ==      !=      <>      +=
-=      *=      /=      //=     %=      **=     &=      |=      ^=      >>=     <<=
```

Следующие лексемы играют роль разделителей в выражениях, списках, словарях и в различных частях инструкций:

```
( ) [ ] { } , : . ` = ;
```

Например, знак равенства (=) отделяет имя и значение в операциях присваивания, а запятая (,) используется для разделения аргументов при вызовах функций, элементов в списках и кортежах и так далее. Точка (.) используется в определениях чисел с плавающей точкой, а многоточие (...) – в расширенных операциях со срезами.

Наконец, следующие символы также имеют специальное назначение:

```
‘ “ # \ @
```

Символы $ и ? не имеют специального назначения и не могут присутствовать в программном коде, за исключением строковых литералов.

Строки документирования

Если первой инструкцией в определении модуля, класса или функции является строка, она становится строкой документирования, описывающей соответствующий объект, как показано в следующем примере:

```
def fact(n):
    “Эта функция находит факториал числа”
    if (n <= 1): return 1
        else: return n * fact(n - 1)
```

Строки документирования иногда используются инструментами просмотра программного кода и автоматического создания документации. Эти строки доступны в виде атрибута `__doc__` объектов, например:

```
>>> print fact._ _doc__
Эта функция находит факториал числа
>>>
```

К строкам документирования применяются те же самые правила оформления отступов, что и для других инструкций в определении. Кроме того, строка документирования не может быть вычисляемой или извлекаться из переменной. Строка документирования всегда определяется как строковый литерал.

Декораторы

Определениям функций, методов или классов может предшествовать специальный символ, известный как *декоратор*, цель которого состоит в том, чтобы изменить поведение определения, следующего за ним. Декораторы начинаются с символа @ и должны располагаться в отдельной строке, непосредственно перед соответствующей функцией, методом или классом. Например:

```
class Foo(object):
    @staticmethod
    def bar():
        pass
```

Допускается указывать несколько декораторов, но каждый из них должен находиться в отдельной строке. Например:

```
@foo
@bar
def spam():
    pass
```

Дополнительную информацию о декораторах можно найти в главе 6 «Функции и функциональное программирование» и в главе 7 «Классы и объектно-ориентированное программирование».

Кодировка символов в исходных текстах

Исходные тексты программ на языке Python обычно записываются в стандартной 7-битовой кодировке ASCII. Однако пользователи, работающие в среде с поддержкой Юникода, могут посчитать это ограничение неудобным, особенно когда возникает необходимость в создании большого количества строковых литералов с национальными символами.

Язык Python позволяет записывать программный код в другой кодировке, для чего в первой или во второй строке программы необходимо добавить специальный комментарий, указывающий тип кодировки:

```
#!/usr/bin/env python
# -*- coding: UTF-8 -*-

s = "Jalapeño" # Строка в кавычках записана непосредственно в кодировке UTF-8.
```

При наличии специального комментария `coding:` строковые литералы могут вводиться непосредственно с клавиатуры, с помощью текстового редактора, поддерживающего требуемую кодировку. Однако другие элементы программы на языке Python, включая идентификаторы и зарезервированные слова, должны содержать только символы ASCII.

3

Типы данных и объекты

Все данные, используемые в программах на языке Python, опираются на понятие *объекта*. В число объектов входят такие фундаментальные типы данных, как числа, строки, списки и словари. Однако имеется возможность определять свои собственные объекты в виде классов. Кроме того, большинство объектов неразрывно связаны со структурой программы и внутренними особенностями функционирования интерпретатора. В этой главе описывается внутреннее устройство модели объектов языка Python и приводится обзор встроенных типов данных. Дальнейшее описание операторов и выражений приводится в главе 4 «Операторы и выражения». Порядок создания пользовательских объектов описывается в главе 7 «Классы и объектно-ориентированное программирование».

Терминология

Любой элемент данных, используемый в программе на языке Python, является объектом. Каждый объект имеет свою идентичность, тип (или класс) и значение. Например, когда в программе встречается инструкция `a = 42`, интерпретатор создает целочисленный объект со значением `42`. Вы можете рассматривать *идентичность* объекта как указатель на область памяти, где находится объект, а идентификатор `a` – как имя, ссылающееся на эту область памяти.

Тип, или *класс* объекта определяет его внутреннее представление, а также методы и операции, которые им поддерживаются. Объект определенного типа иногда называют *экземпляром* этого типа. После создания объекта его идентичность и тип не могут быть изменены. Если значение объекта может изменяться, такой объект называют *изменяемым*. Если значение не может быть изменено, такой объект называют *неизменяемым*. Объект, содержащий ссылки на другие объекты, называется *контейнером*, или *коллекцией*.

Большинство объектов обладают определенным количеством атрибутов и методов. *Атрибут* – это значение, связанное с объектом. *Метод* – это

функция, выполняющая некоторую операцию над объектом, когда вызывается, как функция. Обращение к атрибутам и методам выполняется с помощью оператора точки (.), как показано в следующем примере:

```
a = 3 + 4j    # Создается комплексное число
r = a.real    # Извлекается действительная часть (атрибут)

b = [1, 2, 3] # Создается список
b.append(7)   # С помощью метода append в конец списка
              # добавляется новый элемент
```

Идентичность и тип объекта

Идентичность объекта, в виде целого числа, можно получить с помощью встроенной функции `id()`. Обычно это целое число соответствует адресу области памяти, где находится объект, однако это является лишь особенностью некоторых реализаций Python, поэтому не следует делать никаких предположений о природе происхождения идентичности. Оператор `is` сравнивает идентичности двух объектов. Встроенная функция `type()` возвращает тип объекта. Ниже приводятся примеры различных способов сравнивания двух объектов:

```
# Сравнение двух объектов
def compare(a,b):
    if a is b:
        # a и b ссылаются на один и тот же объект
        инструкции
    if a == b:
        # объекты a и b имеют одинаковые значения
        инструкции
    if type(a) is type(b):
        # объекты a и b имеют один и тот же тип
        инструкции
```

Тип объекта сам по себе является объектом, который называется классом объекта. Этот объект уникален и всегда один и тот же для всех экземпляров данного типа. По этой причине типы можно сравнивать с помощью оператора `is`. Все объекты типов имеют имена, которые можно использовать при проверке типа. Большинство этих имен, такие как `list`, `dict` и `file`, являются встроенными. Например:

```
if type(s) is list:
    s.append(item)

if type(d) is dict:
    d.update(t)
```

Поскольку типы могут расширяться за счет определения классов, проверку типов лучше всего выполнять с помощью встроенной функции `isinstance(`*object*`, `*type*`)`. Например:

```
if isinstance(s,list):
    s.append(item)
```

```
if isinstance(d,dict):
    d.update(t)
```

Функция `isinstance()` учитывает возможность наследования, поэтому она является предпочтительным способом проверки типа любого объекта в языке Python.

Хотя проверка типов объектов является легкодоступной операцией, тем не менее она не так полезна, как можно было бы подумать. Во-первых, излишние проверки ведут к снижению производительности программы. Во-вторых, в программах не всегда определяются объекты, которые четко укладываются в иерархию наследования. Например, цель проверки `isinstance(s,list)` в предыдущем примере состоит в том, чтобы убедиться, что объект «напоминает список», но этот прием не годится для проверки объектов, которые имеют такой же программный интерфейс, что и списки, но напрямую не наследуют встроенный тип `list`. С другой стороны, проверка типов объектов может пригодиться в случае, если в программе определяются абстрактные базовые классы. Подробнее об этом рассказывается в главе 7.

Подсчет ссылок и сборка мусора

Для всех объектов в программе ведется подсчет ссылок. Счетчик ссылок на объект увеличивается всякий раз, когда ссылка на объект записывается в новую переменную или когда объект помещается в контейнер, такой как список, кортеж или словарь, как показано ниже:

```
a = 37      # Создается объект со значением 37
b = a       # Увеличивает счетчик ссылок на объект 37
c = []
c.append(b) # Увеличивает счетчик ссылок на объект 37
```

В этом примере создается единственный объект, содержащий значение `37`. `a`, – это всего лишь имя, ссылающееся на созданный объект. Когда переменная `a` присваивается переменной `b`, `b` становится еще одним именем того же самого объекта, при этом счетчик ссылок на объект увеличивается на 1. Точно так же, когда переменная `b` помещается в список, счетчик ссылок увеличивается на единицу. На протяжении всего примера существует только один объект, содержащий значение `37`. Все остальные операции просто приводят к созданию новых ссылок на него.

Счетчик ссылок уменьшается, когда вызывается инструкция `del` или когда поток выполнения покидает область видимости ссылки (или при присваивании другого значения). Например:

```
del a      # Уменьшает счетчик ссылок на объект 37
b = 42     # Уменьшает счетчик ссылок на объект 37
c[0] = 2.0 # Уменьшает счетчик ссылок на объект 37
```

Определить текущее количество ссылок на объект можно с помощью функции `sys.getrefcount()`. Например:

```
>>> a = 37
>>> import sys
```

```
>>> sys.getrefcount(a)
7
>>>
```

Во многих случаях количество ссылок оказывается намного больше, чем можно было бы предположить. Для неизменяемых типов данных, таких как числа и строки, интерпретатор весьма активно стремится использовать в разных частях программы один и тот же объект, чтобы уменьшить объем потребляемой памяти.

Когда счетчик ссылок на объект достигает нуля, он уничтожается сборщиком мусора. Однако в некоторых случаях могут возникать циклические зависимости между коллекциями объектов, которые больше не используются. Например:

```
a = { }
b = { }
a['b'] = b     # a содержит ссылку на b
b['a'] = a     # b содержит ссылку на a
del a
del b
```

В этом примере инструкция `del` уменьшает счетчики ссылок на объекты `a` и `b` и уничтожает имена, ссылавшиеся на объекты в памяти. Однако поскольку объекты содержат ссылки друг на друга, счетчики ссылок не достигают нуля и объекты продолжают существовать (что приводит к утечке памяти). Чтобы исправить эту проблему, интерпретатор периодически выполняет проверку наличия циклических зависимостей, отыскивает такие недоступные объекты и удаляет их. Механизм поиска циклических зависимостей запускается периодически, как только интерпретатор обнаруживает, что в процессе выполнения программы объем занимаемой памяти начинает расти. Точное поведение механизма может корректироваться и управляться с помощью функций из модуля `gc` (см. главу 13, «Службы Python времени выполнения»).

Ссылки и копии

Когда в программе выполняется присваивание, такое как `a = b`, создается новая ссылка на `b`. Для неизменяемых объектов, таких как числа или строки, такое присваивание фактически создает копию объекта `b`. Однако для изменяемых объектов, таких как списки или словари, присваивание выполняется совершенно иначе. Например:

```
>>> a = [1,2,3,4]
>>> b = a          # b - это ссылка на a
>>> b is a
True
>>> b[2] = -100    # Изменение элемента списка b
>>> a              # Обратите внимание, что список a также изменился
[1, 2, -100, 4]
>>>
```

В этом примере переменные a и b ссылаются на один и тот же объект, поэтому изменения, произведенные в одной переменной, можно наблюдать в другой. Чтобы избежать этого, необходимо вместо новой ссылки создать копию самого объекта.

К контейнерным объектам, таким как списки или словари, можно применить два вида копирования: поверхностное копирование и глубокое копирование. При *поверхностном копировании* создается новый объект, но он будет заполнен ссылками на элементы, которые содержались в оригинале. Например:

```
>>> a = [ 1, 2, [3,4] ]
>>> b = list(a)         # Создание поверхностной копии списка a.
>>> b is a
False
>>> b.append(100)       # Добавление нового элемента в список b.
>>> b
[1, 2, [3, 4], 100]
>>> a                   # Обратите внимание, что список a не изменился
[1, 2, [3, 4]]
>>> b[2] [0] = -100     # Изменение элемента в списке b
>>> b
[1, 2, [-100, 4], 100]
>>> a                   # Обратите внимание, что изменился и список a
[1, 2, [-100, 4]]
>>>
```

В данном случае a и b представляют два независимых списка, но элементы, содержащиеся в них, являются общими. Поэтому изменение одного из элементов в списке a приводит к изменению элемента в списке b.

При *глубоком копировании* создается новый объект и рекурсивно создаются копии всех объектов, содержащихся в оригинале. В языке Python нет встроенной функции, выполняющей глубокое копирование объектов. Однако в стандартной библиотеке существует функция copy.deepcopy(), пример использования которой приводится ниже:

```
>>> import copy
>>> a = [1, 2, [3, 4]]
>>> b = copy.deepcopy(a)
>>> b[2] [0] = -100
>>> b
[1, 2, [-100, 4]]
>>> a               # Обратите внимание, что список a не изменился
[1, 2, [3, 4]]
>>>
```

Объекты первого класса

Все объекты в языке Python могут быть отнесены к объектам первого класса. Это означает, что все объекты, имеющие идентификатор, имеют одинаковый статус. Это также означает, что все объекты, имеющие идентифика-

тор, можно интерпретировать, как данные. Например, ниже приводится простой словарь, содержащий два значения:

```
items = {
    'number' : 42
    'text' : "Hello World"
}
```

«Первоклассную» природу объектов можно наблюдать, если попробовать добавить в этот словарь несколько необычных элементов. Например:

```
items["func"] = abs           # Добавляется функция abs()
import math
items["mod"] = math           # Добавляется модуль
items["error"] = ValueError   # Добавляется тип исключения
nums = [1,2,3,4]
items["append"] = nums.append # Добавляется метод другого объекта
```

В этом примере в словарь были добавлены функция, модуль, исключение и метод другого объекта. При желании можно обратиться к элементам словаря вместо оригинальных объектов, и такой код будет работать. Например:

```
>>> items["func"](-45)          # Вызовет функцию abs(-45)
45
>>> items["mod"].sqrt(4)        # Вызовет функцию math.sqrt(4)
2.0
>>> try:
...     x = int("a lot")
... except items["error"] as e: # Идентична инструкции except ValueError as e
...     print("Couldn't convert")
...
Couldn't convert
>>> items["append"](100)        # Вызовет метод nums.append(100)
>>> nums
[1, 2, 3, 4, 100]
>>>
```

Тот факт, что все объекты в языке Python **являются объектами первого класса**, зачастую недооценивается начинающими программистами, тогда как это позволяет писать очень компактные и гибкие программы. Например, предположим, что имеется текстовая строка, такая как "GOOG,100,490.10", и вам необходимо превратить ее в список полей, попутно выполнив необходимые преобразования типов. Ниже приводится грамотный способ реализации такого преобразования: создается список типов (которые также являются объектами первого класса) и выполняется несколько простых операций по обработке списка:

```
>>> line = "GOOG,100,490.10"
>>> field_types = [str, int, float]
>>> raw_fields = line.split(',')
>>> fields = [ty(val) for ty,val in zip(field_types,raw_fields)]
>>> fields
['GOOG', 100, 490.10000000000002]
>>>
```

Встроенные типы представления данных

В языке **Python существует около десятка встроенных типов данных, которые** используются для представления большинства данных в программах. Эти типы сгруппированы в несколько основных категорий, как показано в табл. 3.1. В колонке «Имя типа» перечислены имена и выражения, которые можно использовать для проверки принадлежности к типу с помощью таких функций, как `isinstance()`. Некоторые типы данных присутствуют только в версии **Python 2, – они отмечены соответствующим образом (в Python 3** эти типы не рекомендуются к использованию или были поглощены другими типами**).**

Таблица 3.1. Встроенные типы представления данных

Категория типов	Имя типа	Описание
None	`type(None)`	Пустой объект `None`.
Числа	`int`	Целые числа
	`long`	Целые числа произвольной точности (только в Python 2)
	`float`	Числа с плавающей точкой
	`complex`	Комплексные числа
	`bool`	Логические значения (`True` и `False`)
Последовательности	`str`	Строки символов
	`unicode`	Строки символов Юникода (только в Python 2)
	`list`	Списки
	`tuple`	Кортежи
	`xrange`	Диапазоны целых чисел, создаваемые функцией `xrange()` (в версии Python 3 называется `range`)
Отображения	`dict`	Словари
Множества	`set`	Изменяемые множества
	`frozenset`	Неизменяемые множества

Тип None

Тип `None` используется для представления пустых объектов (объектов, не имеющих значений). В языке **Python существует ровно один пустой объ**ект, который в программах записывается как `None`. Этот объект возвращается функциями, которые не имеют явно возвращаемого значения. Объект `None` часто используется как значение по умолчанию для необязательных аргументов, благодаря чему функция может определить, были ли переданы фактические значения в таких аргументах. Объект `None` не имеет атрибутов и в логическом контексте оценивается как значение `False`.

Числовые типы

В языке Python используется пять числовых типов: логический, целые числа, длинные целые числа, числа с плавающей точкой и комплексные числа. За исключением логических значений, все остальные числовые объекты представляют числа со знаком. Все числовые типы относятся к разряду неизменяемых.

Логический тип представлен двумя значениями: `True` и `False`. Имена `True` и `False` отображаются в числовые значения 1 и 0 соответственно. Целочисленный тип представляет диапазон чисел от –2147483648 до 2147483647 (на некоторых аппаратных архитектурах этот диапазон может быть шире). Тип длинных целых чисел способен представлять числа неограниченной величины (ограничение накладывается только объемом доступной памяти). Даже при том, что существует два целочисленных типа, тем не менее интерпретатор Python старается скрыть имеющиеся различия (фактически, в Python 3 эти два типа были объединены в один). Поэтому, если вам доведется увидеть в программном коде упоминание о типе длинных целых чисел, в большинстве случаев это свидетельствует лишь об особенностях реализации, которые можно игнорировать, – просто используйте целочисленный тип во всех операциях над целыми числами. Единственное исключение составляет случай проверки принадлежности значения к целочисленному типу. В Python 2 выражение `isinstance(x, int)` вернет `False`, если `x` является длинным целым числом.

Числа с плавающей точкой в языке Python представлены обычными числами с плавающей точкой двойной точности (64 бита). Как правило, это представление соответствует стандарту IEEE 754, который позволяет обеспечивать представление примерно 17 значимых разрядов, с экспонентой в диапазоне от –308 до 308. Это полностью соответствует типу `double` в языке C. Язык Python не поддерживает 32-битные числа с плавающей точкой одинарной точности. Если в программе потребуется иметь возможность более точно управлять точностью представления чисел и объемом занимаемой ими памяти, можно воспользоваться расширением `numpy` (которое доступно по адресу: *http://numpy.sourceforge.net*).

Комплексные числа представлены парами чисел с плавающей точкой. Действительная и мнимая части комплексного числа `z` доступны как атрибуты `z.real` и `z.imag`. Метод `z.conjugate()` возвращает сопряженное значение комплексного числа `z` (сопряженным значением для `a+bj` будет `a-bj`).

Числовые типы обладают свойствами и методами, предназначенными для упрощения арифметических операций над смешанными типами чисел. Для совместимости с рациональными числами (этот тип определен в модуле `fractions`) целые числа имеют свойства `numerator` и `denominator`. Для совместимости с комплексными числами целые числа и числа с плавающей точкой имеют свойства `real` и `imag`, а также метод `conjugate()`. Число с плавающей точкой может быть преобразовано в пару целых чисел, представляющих дробь, с помощью метода `as_integer_ratio()`. Метод `is_integer()` проверяет, можно ли представить число с плавающей точкой как целое

значение. Методы `hex()` и `fromhex()` могут использоваться для обработки чисел с плавающей точкой в низкоуровневом двоичном представлении.

В библиотечных модулях определены некоторые дополнительные числовые типы. Модуль `decimal` обеспечивает поддержку обобщенной десятичной арифметики. Модуль `fractions` определяет тип рациональных чисел. Подробнее эти модули рассматриваются в главе 14 «Математика».

Последовательности

Последовательности используются для представления упорядоченных множеств объектов, элементы которых индексируются целыми положительными числами. К этой категории относятся строки, списки и кортежи. Строки – это последовательности символов, а списки и кортежи – это последовательности произвольных объектов. Строки и кортежи относятся к разряду неизменяемых объектов; списки допускают возможность добавления, удаления и изменения своих элементов. Все последовательности поддерживают итерации.

Операции, общие для всех типов последовательностей

В табл. 3.2 перечислены операторы и методы, которые могут применяться к любым типам последовательностей. Получить доступ к *i*-му элементу последовательности *s* можно с помощью оператора индексирования *s*[*i*]. Выбрать фрагмент последовательности можно с помощью оператора среза *s*[*i*:*j*] или его расширенной версии *s*[*i*:*j*:*stride*] (эти операторы описываются в главе 4). Длину любой последовательности можно определить с помощью встроенной функции `len(`*s*`)`. Отыскать минимальное и максимальное значения в последовательности можно с помощью встроенных функций `min(`*s*`)` и `max(`*s*`)`. Однако эти функции могут действовать только с последовательностями, содержащими элементы, допускающие возможность упорядочения (обычно это строки и числа). Функция `sum(`*s*`)` возвращает сумму элементов последовательности, но она действует только с числовыми данными.

В **табл. 3.3 перечислены дополнительные операторы, которые могут применяться** к изменяемым последовательностям, таким как списки.

Таблица 3.2. Операторы и функции, общие для всех типов последовательностей

Оператор или функция	Описание
s[*i*]	Возвращает *i*-й элемент последовательности *s*
s[*i*:*j*]	Возвращает срез
s[*i*:*j*:*stride*]	Расширенная операция получения среза
`len(`*s*`)`	Возвращает число элементов в последовательности *s*
`min(`*s*`)`	Возвращает минимальное значение в последовательности *s*
`max(`*s*`)`	Возвращает максимальное значение в последовательности *s*

Таблица 3.2 (продолжение)

Оператор или функция	Описание
`sum(s [,initial])`	Возвращает сумму элементов последовательности *s*
`all(s)`	Проверяет, оцениваются ли все элементы последовательности *s*, как `True`
`any(s)`	Проверяет, оценивается ли хотя бы один элемент последовательности *s*, как `True`

Таблица 3.3. Операторы, применяемые к изменяемым последовательностям

Оператор	Описание
`s[i] = v`	Присваивает значения элементу
`s[i:j] = t`	Присваивает последовательность срезу
`s[i:j:stride] = t`	Расширенная операция присваивания срезу
`del s[i]`	Удаляет элемент
`del s[i:j]`	Удаляет срез
`del s[i:j:stride]`	Расширенная операция удаления среза

Списки

Методы, поддерживаемые списками, перечислены в табл. 3.4. Встроенная функция `list(s)` преобразует любой итерируемый объект в список. Если аргумент *s* уже является списком, эта функция создает новый список, используя операцию поверхностного копирования. Метод `s.append(x)` добавляет в конец списка новый элемент *x*. Метод `s.index(x)` отыскивает в списке первое вхождение элемента *x*. В случае отсутствия искомого элемента возбуждается исключение `ValueError`. Точно так же метод `s.remove(x)` удаляет из списка первое вхождение элемента *x* или возбуждает исключение `ValueError`, если искомый элемент отсутствует в списке. Метод `s.extend(t)` расширяет список *s*, добавляя в конец элементы последовательности *t*.

Метод `s.sort()` сортирует элементы списка и может принимать функцию вычисления ключа и признак сортировки в обратном порядке – оба эти аргумента должен передаваться как именованные аргументы. Функция вычисления ключа применяется к каждому элементу списка перед выполнением операции сравнения в процессе сортировки. Эта функция должна принимать единственный элемент и возвращать значение, которое будет использоваться при сравнении. Функцию вычисления ключа удобно использовать, когда требуется реализовать особые виды сортировки, такие как сортировка списка строк без учета регистра символов. Метод `s.reverse()` изменяет порядок следования элементов в списке на обратный. Оба метода, `sort()` и `reverse()`, изменяют сам список и возвращают `None`.

Таблица 3.4. Методы списков

Метод	Описание
`list(s)`	Преобразует объект *s* в список.
`s.append(x)`	Добавляет новый элемент *x* в конец списка *s*.
`s.extend(t)`	Добавляет новый список *t* в конец списка *s*.
`s.count(x)`	Определяет количество вхождений *x* в список *s*.
`s.index(x [,start [,stop]])`	Возвращает наименьшее значение индекса *i*, где *s*[*i*] == *x*. *Необязательные значения* `start` *и* `stop` *определяют индексы начального и конечного элементов диапазона, где выполняется поиск.*
`s.insert(i,x)`	Вставляет *x* в элемент с индексом *i*.
`s.pop([i])`	Возвращает *i*-й элемент и удаляет его из списка. Если индекс *i* не указан, возвращается последний элемент.
`s.remove(x)`	Отыскивает в списке *s* элемент со значением *x* и удаляет его.
`s.reverse()`	Изменяет порядок следования элементов в списке *s* на обратный.
`s.sort([key [, reverse]])`	Сортирует элементы списка *s*. *key* – это функция, которая вычисляет значение ключа. *reverse* – признак сортировки в обратном порядке. Аргументы *key* и *reverse* всегда должны передаваться как именованные аргументы.

Строки

В языке Python 2 имеется два типа строковых объектов. Байтовые строки – это последовательности байтов, содержащих 8-битные данные. Они могут содержать двоичные данные и байты со значением NULL. Строки Юникода – это последовательности декодированных символов Юникода, которые внутри представлены 16-битными целыми числами. Это позволяет определить до 65 536 различных символов. Стандарт Юникода предусматривает поддержку до 1 миллиона отдельных символов, тем не менее эти дополнительные символы не поддерживаются в языке Python по умолчанию. Они кодируются, как специальные двухсимвольные (4-байтовые) последовательности, именуемые *суррогатной парой*; обязанность по их интерпретации возлагается на само приложение. В качестве дополнительной возможности язык Python позволяет хранить символы Юникода в виде 32-битных целых чисел. Когда эта возможность включена, Python позволяет представлять полный диапазон значений Юникода от `U+000000` до `U+110000`. Все функции, работающие со строками Юникода, поддерживают эту особенность.

Строки поддерживают методы, перечисленные в табл. 3.5. Все эти методы оперируют экземплярами строк, однако ни один из них в действительно-

сти не изменяет сами строки, находящиеся в памяти. То есть такие методы, как *s*.capitalize(), *s*.center() и *s*.expandtabs(), никогда не модифицируют строку *s* и всегда возвращают новую строку. Методы проверки символов, такие как *s*.isalnum() и *s*.isupper(), возвращают True или False, если все символы в строке удовлетворяют критерию проверки. Кроме того, эти функции всегда возвращают значение False для строк с нулевой длиной.

Методы *s*.find(), *s*.index(), *s*.rfind() и *s*.rindex() используются для поиска подстроки. Все эти методы возвращают целочисленный индекс подстроки в строке *s*. Кроме того, если подстрока не будет найдена, метод find() возвращает -1, тогда как метод index() возбуждает исключение ValueError. Метод *s*.replace() используется для замены подстроки другой подстрокой. Важно отметить, что все эти методы работают только с простыми подстроками. Для поиска с помощью регулярных выражений используются функции из библиотечного модуля re.

Методы *s*.split() и *s*.rsplit() разбивают строку по указанному разделителю, создавая список полей. Методы *s*.partition() и *s*.rpartition() отыскивают подстроку-разделитель и разбивают исходную строку *s* на три части: текст до разделителя, строка-разделитель и текст после разделителя.

Многие строковые методы принимают необязательные аргументы *start* и *end*, которые должны быть целочисленными значениями, определяющими начальный и конечный индексы в строке *s*. В большинстве случаев эти аргументы могут принимать отрицательные значения, в этом случае отсчет индексов начинается с конца строки.

Метод *s*.translate() обеспечивает улучшенную возможность подстановки символов, например, для быстрого удаления из строки всех управляющих символов. В качестве аргумента он принимает таблицу замены, содержащую отображение «один-к-одному» символов оригинальной строки в символы результата. Для 8-битных строк таблица замены представляет собой 256-символьную строку. Для строк Юникода в качестве таблицы можно использовать любую последовательность *s*, для которой выражение *s*[*n*] возвращает целочисленный код символа или символ Юникода, соответствующий символу Юникода с целочисленным кодом *n*.

Методы *s*.encode() и *s*.decode() используются для преобразования строковых данных *в* или *из* указанной кодировки и принимают название кодировки, например: 'ascii', 'utf-8' или 'utf-16'. Чаще всего эти методы используются для преобразования строк Юникода в данные с кодировкой, подходящей для операций ввода-вывода и подробнее описываются в главе 9 «Ввод и вывод». Не забывайте, что в Python 3 метод encode() доступен только для строк, а метод decode() – только для данных типа bytes.

Метод *s*.format() используется для форматирования строк. Он принимает комбинацию позиционных и именованных аргументов. Символы подстановки в строке *s* обозначаются как {*item*} и замещаются соответствующими аргументами. Ссылки на позиционные аргументы обозначаются как {0}, {1} и т. д. Ссылки на именованные аргументы обозначаются как {*name*}, где *name* – имя аргумента. Например:

```
>>> a = "Вас зовут {0} и вам {age} лет"
>>> a.format("Майк", age=40)
'Вас зовут Майк и вам 40 лет'
>>>
```

Символы подстановки {*item*} в строках формата могут также включать простые индексы и атрибуты поиска. Символ подстановки {*item*[*n*]}, где *n* – это число, замещается *n*-м элементом последовательности *item*. Символ подстановки {*item*[*key*]}, где *key* – это строка, замещается элементом словаря *item*["*key*"]. Символ подстановки {*item.attr*} замещается значением атрибута *attr* объекта *item*. Дополнительные подробности о методе `format()` приводятся в разделе «Форматирование строк» главы 4.

Таблица 3.5. Строковые методы

Метод	Описание
`s.capitalize()`	Преобразует первый символ в верхний регистр.
`s.center(width [, pad])`	Центрирует строку в поле шириной *width*. Аргумент *pad* определяет символ, которым оформляются отступы слева и справа.
`s.count(sub [,start [,end]])`	Подсчитывает число вхождений заданной подстроки *sub*.
`s.decode([encoding [,errors]])`	Декодирует строку и возвращает строку Юникода (только для байтовых строк).
`s.encode([encoding [,errors]])`	Возвращает кодированную версию строки (только для строк Юникода).
`s.endswith(suffix [,start [,end]])`	Проверяет, оканчивается ли строка подстрокой *suffix*.
`s.expandtabs([tabsize])`	Замещает символы табуляции пробелами.
`s.find(sub [, start [,end]])`	Отыскивает первое вхождение подстроки *sub* или возвращает -1.
`s.format(*args, **kwargs)`	Форматирует строку *s*.
`s.index(sub [, start [,end]])`	Отыскивает первое вхождение подстроки *sub* или возбуждает исключение.
`s.isalnum()`	Проверяет, являются ли все символы в строке алфавитно-цифровыми символами.
`s.isalpha()`	Проверяет, являются ли все символы в строке алфавитными символами.
`s.isdigit()`	Проверяет, являются ли все символы в строке цифровыми символами.
`s.islower()`	Проверяет, являются ли все символы в строке символами нижнего регистра.
`s.isspace()`	Проверяет, являются ли все символы в строке пробельными символами.

Таблица 3.5 (продолжение)

Метод	Описание
s.istitle()	Проверяет, являются ли первые символы всех слов символами верхнего регистра.
s.isupper()	Проверяет, являются ли все символы в строке символами верхнего регистра.
s.join(*t*)	Объединяет все строки, находящиеся в последовательности *t*, используя s как строку-разделитель.
s.ljust(*width* [, *fill*])	Выравнивает строку *s* по левому краю в поле шириной *width*.
s.lower()	Преобразует символы строки в нижний регистр.
s.lstrip([*chrs*])	Удаляет начальные пробельные символы или символы, перечисленные в аргументе *chrs*.
s.partition(*sep*)	Разбивает строку по подстроке-разделителю *sep*. Возвращает кортеж (*head*,*sep*,*tail*) или (*s*, "","") , если подстрока *sep* отсутствует в строке *s*.
s.replace(*old*, *new* [,*maxreplace*])	Замещает подстроку *old* подстрокой *new*.
s.rfind(*sub* [,*start* [,*end*]])	Отыскивает последнее вхождение подстроки.
s.rindex(*sub* [,*start* [,*end*]])	Отыскивает последнее вхождение подстроки или возбуждает исключение.
s.rjust(*width* [, *fill*])	Выравнивает строку *s* по правому краю в поле шириной *width*.
s.rpartition(*sep*)	Разбивает строку по подстроке-разделителю *sep*, *но поиск выполняется с конца строки.*
s.rsplit([*sep* [,*maxsplit*]])	Разбивает строку, начиная с конца, по подстроке-разделителю *sep*. Аргумент *maxsplit* определяет максимальное число разбиений. Если аргумент *maxsplit* не указан, идентичен методу split().
s.rstrip([*chrs*])	Удаляет конечные пробельные символы или символы, перечисленные в аргументе *chrs*.
s.split([*sep* [,*maxsplit*]])	Разбивает строку по подстроке-разделителю *sep*. Аргумент *maxsplit* определяет максимальное число разбиений.
s.splitlines([*keepends*])	Преобразует строку в список строк. Если аргумент *keepnds* имеет значение 1, завершающие символы перевода строки остаются нетронутыми.

Метод	Описание
s.startswith(*prefix* [,*start* [,*end*]])	Проверяет, начинается ли строка подстрокой *prefix*.
s.strip([*chrs*])	Удаляет начальные и конечные пробельные символы или символы, перечисленные в аргументе *chrs*.
s.swapcase()	Приводит символы верхнего регистра к нижнему, и наоборот.
s.title()	Возвращает версию строки, в которой первые символы всех слов приведены к верхнему регистру.
s.translate(*table* [,*deletechars*])	Выполняет преобразование строки в соответствии с таблицей замены *table*, *удаляет символы, перечисленные в аргументе deletechars*.
s.upper()	Преобразует символы строки в верхний регистр.
s.zfill(*width*)	Дополняет строку нулями слева до достижения ею длины *width*.

Объекты xrange()

Встроенная функция xrange([*i*,]*j* [,*stride*]) создает объект, представляющий диапазон целых чисел *k*, где *i* <= *k* < *j*. Первый индекс *i* и шаг *stride* по умолчанию являются необязательными и имеют значения 0 и 1 соответственно. Объект xrange вычисляет свои значения в момент обращения к нему, и хотя он выглядит, как последовательность, тем не менее он имеет ряд ограничений. Например, он не поддерживает ни один из стандартных операторов среза. Область применения объекта xrange ограничена простыми циклами.

Следует отметить, что в Python 3 функция xrange() была переименована в range(). Однако она действует точно так же, как описано выше.

Отображения

Объекты отображений представляют произвольные коллекции объектов, которые могут индексироваться другими коллекциями практически произвольных ключей. В отличие от последовательностей, отображения являются неупорядоченными коллекциями и могут индексироваться числами, строками и другими объектами. Отображения относятся к разряду изменяемых коллекций.

Единственным встроенным типом отображений являются словари, которые представляют собой разновидность таблиц хешей, или ассоциативных массивов. В качестве ключей словаря допускается использовать любые неизменяемые объекты (строки, числа, кортежи и так далее). Списки, слова-

ри и кортежи, содержащие изменяемые объекты, не могут использоваться в качестве ключей (словари требуют, чтобы значения ключей оставались постоянными).

Для выборки элементов из объекта отображения используется оператор индексирования по ключу *m*[*k*], где *k* — значение ключа. Если требуемый ключ отсутствует в словаре, возбуждается исключение KeyError. Функция len(*m*) возвращает количество элементов, хранящихся в отображении. Методы и операторы, применимые к словарям, перечислены в табл. 3.6.

Таблица 3.6. Методы и операторы, поддерживаемые словарями

Оператор или метод	Описание
len(*m*)	Возвращает количество элементов в словаре *m*.
m[*k*]	Возвращает элемент словаря *m* с ключом *k*.
m[*k*]=*x*	Записывает в элемент *m*[*k*] значение *x*.
del *m*[*k*]	Удаляет элемент *m*[*k*].
k in *m*	Возвращает True, если ключ *k* присутствует в словаре *m*.
m.clear()	Удаляет все элементы из словаря *m*.
m.copy()	Создает копию словаря *m*.
m.fromkeys(*s* [,*value*])	Создает новый словарь с ключами, перечисленными в последовательности *s*, а все значения устанавливает равными *value*.
m.get(*k* [,*v*])	Возвращает элемент *m*[*k*], если таковой имеется, в противном случае возвращает *v*.
m.has_key(*k*)	Возвращает True, если в словаре *m* имеется ключ *k*, в противном случае возвращает False. (Вместо этого метода рекомендуется использовать оператор in. Доступен только в Python 2.)
m.items()	Возвращает последовательность пар (*key*, *value*).
m.keys()	Возвращает последовательность ключей.
m.pop(*k* [,*default*])	Возвращает элемент *m*[*k*], если таковой имеется, и удаляет его из словаря; в противном случае возвращает *default*, если этот аргумент указан, или возбуждает исключение KeyError.
m.popitem()	Удаляет из словаря случайную пару (*key*, *value*) и возвращает ее в виде кортежа.
m.setdefault(*k* [, *v*])	Возвращает элемент *m*[*k*], если таковой имеется, в противном случае возвращает значение *v* и создает новый элемент словаря *m*[*k*] = *v*.
m.update(*b*)	Добавляет все объекты из *b* в словарь *m*.
m.values()	Возвращает последовательность всех значений в словаре *m*.

Большинство методов, перечисленных в табл. 3.6, используются для извлечения и манипулирования содержимым словарей. Метод *m*.clear() удаляет все элементы. Метод *m*.update(*b*) добавляет в текущее отображение все пары (*key*,*value*), присутствующие в отображении *b*. Метод *m*.get(*k* [,*v*]) извлекает объект и позволяет указывать необязательное значение по умолчанию *v*, которое будет возвращаться в случае отсутствия в словаре указанного ключа. Метод *m*.setdefault(*k* [,*v*]) напоминает метод *m*.get(), за исключением того, что он не только вернет значение *v* в случае отсутствия запрошенного ключа, но и создаст элемент *m*[*k*] = *v*. Если значение *v* не указано, по умолчанию будет использоваться значение None. Метод *m*.pop() возвращает элемент и одновременно удаляет его из словаря. Метод *m*.popitem() используется для удаления содержимого словаря в цикле.

Метод *m*.copy() выполняет поверхностное копирование элементов отображения и помещает их в новое отображение. Метод *m*.fromkeys(*s* [,*value*]) создает новое отображение с ключами, содержащимися в последовательности *s*. Полученное отображение будет иметь тот же тип, что и оригинал *m*. В качестве значений для всех элементов будет использовано значение None, если в необязательном аргументе *value* не задано альтернативное значение. Метод fromkeys() определен, как метод класса, поэтому его можно вызывать относительно имени класса, например: dict.fromkeys().

Метод *m*.items() возвращает последовательность, содержащую пары (*key*, *value*). Метод *m*.keys() возвращает последовательность всех ключей, а метод *m*.values() – последовательность всех значений. Единственной безопасной операцией над результатами этих методов являются итерации. В Python 2 результатом этих методов является список, но в Python 3 результатом является итератор, позволяющий выполнить обход текущего содержимого отображения. Если программа будет использовать результаты этих методов как итераторы, она будет совместима с обеими версиями Python. Если потребуется сохранить результаты этих методов как обычные данные, их следует скопировать и сохранить в списке. Например, items = list(m.items()). Если программе достаточно просто получить список всех ключей, лучше использовать инструкцию keys = list(m).

Множества

Множества – это неупорядоченные коллекции уникальных элементов. В отличие от последовательностей, множества не поддерживают операции индексирования и получения срезов. От словарей их отличает отсутствие ключей, связанных с объектами. Элементы, помещаемые в множество, должны быть неизменяемыми. Существует два типа множеств: set – изменяемое множество, и frozenset – неизменяемое множество. Оба типа множеств создаются с помощью пары встроенных функций:

```
s = set([1,5,10,15])
f = frozenset(['a',37,'hello'])
```

Обе функции, set() и frozenset(), заполняют создаваемое множество, выполняя итерации по значениям, хранящимся в аргументе. Оба типа множеств поддерживают методы, перечисленные в табл. 3.7.

Таблица 3.7. Методы и операторы, поддерживаемые множествами

Метод	Описание
`len(s)`	Возвращает количество элементов в множестве *s*.
`s.copy()`	Создает копию множества *s*.
`s.difference(t)`	Разность множеств. Возвращает все элементы из множества *s*, отсутствующие в *t*.
`s.intersection(t)`	Пересечение множеств. Возвращает все элементы, присутствующие в обоих множествах *s* и *t*.
`s.isdisjoint(t)`	Возвращает `True`, если множества *s* и *t* не имеют общих элементов.
`s.issubset(t)`	Возвращает `True`, если множество *s* является подмножеством *t*.
`s.issuperset(t)`	Возвращает `True`, если множество *s* является надмножеством *t*.
`s.symmetric_difference(t)`	Симметричная разность множеств. Возвращает все элементы, которые присутствуют в множестве *s* или *t*, но не в обоих сразу.
`s.union(t)`	Объединение множеств. Возвращает все элементы, присутствующие в множестве *s* или *t*.

Методы `s.difference(t)`, `s.intersection(t)`, `s.symmetric_difference(t)` и `s.union(t)` реализуют стандартные математические операции над множествами. Возвращаемое значение имеет тот же тип, что и множество *s* (`set` или `frozenset`). Аргумент *t* может быть любым объектом языка Python, поддерживающим итерации. В число таких объектов входят множества, списки, кортежи и строки. Эти операции над множествами доступны также в виде математических операторов, которые подробнее описываются в главе 4.

Дополнительно изменяемые множества (тип `set`) поддерживают методы, перечисленные в табл. 3.8.

Таблица 3.8. Методы, поддерживаемые изменяемыми множествами

Метод	Описание
`s.add(item)`	Добавляет элемент *item* в *s*. Ничего не делает, если этот элемент уже имеется в множестве.
`s.clear()`	Удаляет все элементы из множества *s*.
`s.difference_update(t)`	Удаляет все элементы из множества *s*, которые присутствуют в *t*.
`s.discard(item)`	Удаляет элемент *item* из множества *s*. Ничего не делает, если этот элемент отсутствует в множестве.
`s.intersection_update(t)`	Находит пересечение *s* и *t* и оставляет результат в *s*.
`s.pop()`	Возвращает произвольный элемент множества и удаляет его из *s*.

Метод	Описание
s.remove(*item*)	Удаляет элемент *item* из *s*. *Если элемент* *item* отсутствует в множестве, возбуждается исключение KeyError.
s.symmetric_difference_update(t)	Находит симметричную разность *s* и *t* и оставляет результат в *s*.
s.update(*t*)	Добавляет все элементы *t* в множество *s*. *t* может быть другим множеством, последовательностью или любым другим объектом, поддерживающим итерации.

Все эти операции изменяют само множество *s*. Аргумент *t* может быть любым объектом, поддерживающим итерации.

Встроенные типы представления структурных элементов программы

Функции, классы и модули в языке Python **являются объектами, которыми** можно манипулировать, как обычными данными. В табл. 3.9 перечислены типы, которые используются для представления различных элементов самих программ.

Таблица 3.9. Встроенные типы Python для представления структурных элементов программ

Категория типов	Имя типа	Описание
Вызываемые	types.BuiltinFunctionType	Встроенные функции и методы
	type	Тип встроенных типов и классов
	object	Родоначальник всех типов и классов
	types.FunctionType	Пользовательские функции
	types.MethodType	Методы классов
Модули	types.ModuleType	Модуль
Классы	object	Родоначальник всех типов и классов
Типы	type	Тип встроенных типов и классов

Обратите внимание, что в табл. 3.9 object и type встречаются дважды, потому что типы и классы, как и функции, являются вызываемыми объектами.

Вызываемые типы

Вызываемые типы представляют объекты, поддерживающие возможность их вызова, как функций. Существует несколько разновидностей объектов, обладающих этим свойством, включая пользовательские функции, встроенные функции, методы экземпляров и классы.

Пользовательские функции

Пользовательские функции являются вызываемыми объектами, которые создаются на уровне модуля с помощью инструкции `def` или оператора `lambda`. Например:

```
def foo(x,y):
    return x + y

bar = lambda x,y: x + y
```

Пользовательская функция *f* обладает следующими атрибутами:

Атрибут	Описание
f.__doc__	Строка документирования
f.__name__	Имя функции
f.__dict__	Словарь, содержащий атрибуты функции
f.__code__	Скомпилированный байт-код функции
f.__defaults__	Кортеж с аргументами по умолчанию
f.__globals__	Словарь, определяющий глобальное пространство имен
f.__closure__	Кортеж, содержащий данные, связанные с вложенными областями видимости

В устаревших версиях Python 2 многие из перечисленных атрибутов имели имена, такие как `func_code`, `func_defaults` и так далее. Перечисленные имена атрибутов совместимы с версиями Python 2.6 и Python 3.

Методы

Методы – это функции, которые определены внутри класса. Существует три основных типа методов: методы экземпляра, методы класса и статические методы:

```
class Foo(object):
    def instance_method(self,arg):
        инструкции
    @classmethod
    def class_method(cls,arg):
        инструкции
    @staticmethod
    def static_method(arg):
        инструкции
```

Метод экземпляра – это метод, который оперирует экземпляром данного класса. Ссылка на экземпляр передается методу в первом аргументе, который в соответствии с соглашениями называется `self`. *Метод класса* оперирует самим классом, как объектом. Объект класса передается методу класса в первом аргументе `cls`. *Статический метод* – это обычная функция,

которая по некоторым соображениям была помещена в класс. Он не получает ссылку ни на экземпляр класса, ни на сам класс.

Методы экземпляра и методы класса представлены специальным типом `types.MethodType`. Однако, чтобы понять, как действует этот специальный тип, необходимо разобраться с тем, как действует механизм поиска атрибутов объекта (`.`). Поиск любого атрибута объекта (`.`) всегда выполняется как отдельная операция, независимо от операции вызова функции. При фактическом вызове метода выполняются обе операции, но в виде отдельных действий. Следующий пример иллюстрирует порядок вызова метода *f*`.instance_method(`*arg*`)` экземпляра класса `Foo` из предыдущего листинга:

```
f = Foo()                # Создает экземпляр
meth = f.instance_method # Отыскивает метод и отмечает отсутствие ()
meth(37)                 # Вызов метода
```

В данном примере `meth` – это *связанный метод*. Связанный метод – это вызываемый объект, который заключает в себе функцию (метод) и экземпляр класса. При вызове связанного метода экземпляр класса передается методу в первом аргументе (`self`). То есть в этом примере `meth` можно рассматривать, как механизм вызова метода, готовый к использованию, который будет запущен при добавлении к нему оператора `()` вызова функции.

Операция поиска метода может также выполняться непосредственно в самом классе. Например:

```
umeth = Foo.instance_method # Поиск instance_method в классе Foo
umeth(f,37)                 # Вызов с явной передачей значения аргумента self
```

В этом примере `umeth` – это *несвязанный метод*. Несвязанный метод – это вызываемый объект, который заключает в себе функцию, но предполагает, что ссылка на экземпляр соответствующего типа будет передаваться в первом аргументе вручную. В данном примере в первом аргументе передается экземпляр `f` класса `Foo`. Если попытаться передать экземпляр неверного типа, будет возбуждено исключение `TypeError`. Например:

```
>>> umeth("hello",5)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: descriptor 'instance_method' requires a 'Foo' object but received a 'str'
(Перевод:
Трассировочная информация (самый последний вызов - самый нижний):
  Файл "<stdin>", строка 1, в <модуль>
TypeError: дескриптор 'instance_method' должен быть объектом класса 'Foo',
получен объект 'str'
)
>>>
```

Для пользовательских классов связанные и несвязанные методы представлены объектами типа `types.MethodType`, которые являются всего лишь тонкой оберткой вокруг обычного объекта функции. Методы поддерживают следующие атрибуты:

Атрибут	Описание
m.__doc__	Строка документирования
m.__name__	Имя метода
m.__class__	Класс, в котором определен данный метод
m.__func__	Объект функции, реализующей данный метод
m.__self__	Ссылка на экземпляр, ассоциированный с данным методом (None – для несвязанных методов)

Одна из важных особенностей **Python 3 состоит в том, что несвязанные методы** больше не обернуты объектом types.MethodType. Если обратиться к методу Foo.instance_method, как показано в предыдущих примерах, он будет интерпретироваться, как обычный объект функции, реализующей метод. Более того, в Python 3 больше не выполняется проверка типа аргумента self.

Встроенные функции и методы

Для представления функций и методов, реализованных на языках C и C++, используется объект types.BuiltinFunctionType. Для встроенных методов доступны следующие атрибуты:

Атрибут	Описание
b.__doc__	Строка документирования
b.__name__	Имя функции/метода
b.__self__	Ссылка на экземпляр, ассоциированный с данным методом (для связанных методов)

Для встроенных функций, таких как len(), аргумент __self__ устанавливается в значение None, что является признаком функции, которая не связана с каким-либо конкретным объектом. Для встроенных методов, таких как *x*.append, где *x* – объект списка, в аргументе __self__ передается ссылка на объект *x*.

Классы и экземпляры как вызываемые объекты

Классы и экземпляры также могут действовать, как вызываемые объекты. Объект класса создается инструкцией class и может вызываться как функция для создания новых экземпляров. В этом случае аргументы вызова передаются методу __init__() класса для инициализации вновь созданного экземпляра. Экземпляр может имитировать поведение функции, если определяет специальный метод __call__(). Если этот метод определен для экземпляра *x*, то инструкция *x*(*args*) вызовет метод *x*.__call__(*args*).

Классы, типы и экземпляры

Когда определяется класс, его объявление обычно создает объект типа `type`. Например:

```
>>> class Foo(object):
...     pass
...
>>> type(Foo)
<type 'type'>
```

В следующей таблице приводятся атрибуты объекта *t* типа `type`, используемые чаще всего:

Атрибут	Описание
t.__doc__	Строка документирования
t.__name__	Имя класса
t.__bases__	Кортеж с базовыми классами
t.__dict__	Словарь, содержащий методы и атрибуты класса
t.__module__	Имя модуля, в котором определен класс
t.__abstractmethods__	Множество имен абстрактных методов (может быть неопределен, если абстрактные методы отсутствуют в классе)

Когда создается экземпляр класса, типом объекта становится класс, определяющий его. Например:

```
>>> f = Foo()
>>> type(f)
<class '__main__.Foo'>
```

В следующей таблице перечислены специальные атрибуты экземпляра *i*:

Атрибут	Описание
i.__class__	Класс, которому принадлежит экземпляр
i.__dict__	Словарь, содержащий данные экземпляра

Атрибут `__dict__` обычно служит для хранения данных, ассоциированных с экземпляром. Когда выполняется операция присваивания, такая как *i.attr = value*, значение *value* сохраняется в этом словаре. Однако, если в пользовательском классе определен атрибут `__slots__`, используется более эффективное представление экземпляра, и у него уже не будет атрибута `__dict__`. Подробнее об объектах и организации объектной модели в языке Python рассказывается в главе 7.

Модули

Тип *модуль* – это контейнер, хранящий объекты, загруженные инструкцией `import`. Например, когда в программе встречается инструкция `import foo`, имя `foo` связывается с соответствующим объектом модуля. Модули формируют свои пространства имен, которые реализуются на основе словарей, доступных в виде атрибута `__dict__`. Всякий раз, когда выполняется обращение к атрибуту модуля (с использованием оператора точки), оно транслируется в операцию поиска по словарю. Например, обращение *m.x* эквивалентно обращению *m.*`__dict__["`*x*`"]`. Точно так же присваивание атрибуту модуля, например: *m.x = y*, эквивалентно присваиванию *m.*`__dict__["`*x*`"] = y`. Модули имеют следующие атрибуты:

Атрибут	Описание
m.`__dict__`	Словарь, содержащий атрибуты модуля
m.`__doc__`	Строка документирования модуля
m.`__name__`	Имя модуля
m.`__file__`	Имя файла, откуда был загружен модуль
m.`__path__`	Полное имя пакета. Определен, только когда объект модуля ссылается на пакет

Встроенные типы данных для внутренних механизмов интерпретатора

Ряд объектов, используемых внутренней реализацией интерпретатора, доступен пользователю. В их число входят объекты с трассировочной информацией, объекты с программным кодом, объекты кадров стека, объекты генераторов, объекты срезов и `Ellipsis`. Все они перечислены в табл. 3.10. Эти объекты достаточно редко напрямую используются в программах, но они могут иметь практическую ценность для разработчиков инструментов и фреймворков.

Таблица 3.10. Встроенные типы данных для внутренних механизмов интерпретатора

Имя типа	Описание
`types.CodeType`	Скомпилированный байт-код
`types.FrameType`	Кадр стека вызовов
`types.GeneratorType`	Объект генератора
`types.TracebackType`	Трассировочная информация для исключения
`slice`	Создается расширенной операцией получения среза
`Ellipsis`	Используется расширенной операцией получения среза

Объекты с программным кодом

Объекты с программным кодом представляют скомпилированный, исполняемый программный код, или *байт-код*, и обычно создаются встроенной функцией `compile()`. Эти объекты похожи на функции, за исключением того, что они не содержат контекста, связанного с пространством имен, в котором этот программный код определен, но при этом они хранят информацию о значениях по умолчанию для аргументов. Объект *c* с программным кодом имеет следующие атрибуты, доступные только для чтения:

Атрибут	Описание
c.co_name	Имя функции
c.co_argcount	Количество позиционных аргументов (включая значения по умолчанию)
c.co_nlocals	Количество локальных переменных, используемых функцией
c.co_varnames	Кортеж с именами локальных переменных
c.co_cellvars	Кортеж с именами переменных, на которые ссылаются вложенные функции
c.co_freevars	Для вложенных функций. Кортеж с именами свободных переменных, которые определены в объемлющих функциях и используются вложенными функциями
c.co_code	Строковое представление байт-кода
c.co_consts	Кортеж литералов, используемых байт-кодом
c.co_names	Кортеж имен, используемых байт-кодом
c.co_filename	Имя файла, в котором был скомпилирован программный код
c.co_firstlineno	Номер первой строки функции
c.co_lnotab	Строка, представляющая отображение смещений в байт-коде в номера строк
c.co_stacksize	Необходимый размер стека (включая место для локальных переменных)
c.co_flags	Целое число, представляющее различные флаги интерпретатора. Бит 2 (0x04) устанавливается, если функция принимает переменное число позиционных аргументов (`*args`). Бит 3 (0x08) устанавливается, если функция принимает переменное число именованных аргументов (`**kwds`). Остальные биты не используются и зарезервированы на будущее

Объекты кадра стека

Объекты кадра стека используются для представления окружения выполнения и в большинстве случаев создаются объектами с трассировочной ин-

формацией (описываются ниже). Объект *f* кадра стека имеет следующие атрибуты, доступные только для чтения:

Атрибут	Описание
f.f_back	Предыдущий кадр стека (кадр стека вызывающей функции)
f.f_code	Текущий выполняемый объект с программным кодом
f.f_locals	Словарь, используемый для поиска локальных переменных
f.f_globals	Словарь, используемый для поиска глобальных переменных
f.f_builtins	Словарь, используемый для поиска встроенных имен
f.f_lineno	Номер строки
f.f_lasti	Текущая инструкция. Представляет индекс в строке байт-кода объекта f_code

Следующие атрибуты могут изменяться (а также использоваться отладчиками и другими инструментами):

Атрибут	Описание
f.f_trace	Функция, которая будет вызываться в начале каждой строки исходного кода
f.f_exc_type	Тип последнего исключения (только в Python 2)
f.f_exc_value	Значение последнего исключения (только в Python 2)
f.f_exc_traceback	Объект с трассировочной информацией последнего исключения (только в Python 2)

Объекты с трассировочной информацией

Объекты с трассировочной информацией создаются в момент появления исключения и содержат информацию о состоянии стека. После входа в блок обработки исключения трассировочную информацию можно получить с помощью функции sys.exc_info(). Объект с трассировочной информацией имеет следующие атрибуты, доступные только для чтения:

Атрибут	Описание
t.tb_next	Ссылка на объект с трассировочной информацией, представляющий следующий уровень (в направлении кадра стека, в котором возникла исключительная ситуация)
t.tb_frame	Ссылка на кадр стека, представляющего текущий уровень
t.tb_lineno	Номер строки, в которой возникла исключительная ситуация
t.tb_lasti	Номер инструкции (на текущем уровне), где возникла исключительная ситуация

Объекты генераторов

Объекты генераторов создаются при вызове функций-генераторов (подробности приводятся в главе 6 «Функции и функциональное программирование»). Функция-генератор создается, если в ней используется специальное ключевое слово `yield`. Объект генератора выступает одновременно в роли итератора и контейнера с информацией о самой функции-генераторе. Объект генератора обладает следующими атрибутами и методами:

Атрибут	Описание
g.gi_code	Объект с программным кодом функции-генератора
g.gi_frame	Кадр стека функции-генератора
g.gi_running	Целое число, указывающее – выполняется ли функция-генератор в настоящий момент
g.next()	Выполняет функцию-генератор, пока не будет встречена следующая инструкция `yield`, и возвращает полученное значение (в Python 3 этот метод вызывает метод `__next__()`)
g.send(*value*)	Передает значение *value* генератору. Это значение возвращается выражением `yield` в функции-генераторе. После этого функция-генератор продолжит выполнение, пока не будет встречена следующая инструкция `yield`. Метод `send()` возвращает значение, полученное от этой инструкции `yield`
g.close()	Закрывает генератор, возбуждая исключение `GeneratorExit` в функции-генераторе. Этот метод вызывается автоматически, когда объект генератора уничтожается сборщиком мусора
g.throw(*exc* [,*exc_value* [,*exc_tb*]])	Возбуждает исключение в функции-генераторе в точке вызова инструкции `yield`. *exc* – тип исключения, `exc_value` – значение исключения и `exc_tb` – необязательный объект с трассировочной информацией. Если исключение перехвачено и обработано, вернет значение, переданное следующей инструкции `yield`

Объекты срезов

Объекты срезов используются для представления срезов, заданных с применением расширенного синтаксиса, например: *a*[*i*:*j*:*stride*], *a*[*i*:*j*, *n*:*m*] или

a[..., *i*:*j*]. Объекты срезов также создаются встроенной функцией slice([*i*,] *j* [,*stride*]). Они обладают следующими атрибутами, доступными только для чтения:

Атрибут	Описание
s.start	Нижняя граница среза. Принимает значение None, если не задана.
s.stop	Верхняя граница среза. Принимает значение None, если не задана.
s.step	Шаг среза. Принимает значение None, если не задан.

Объекты срезов также имеют единственный метод *s*.indices(*length*). Он принимает длину и возвращает кортеж (*start*,*stop*,*stride*) с параметрами среза для последовательности с указанной длиной. Например:

```
s = slice(10,20) # Объект среза [10:20]
s.indices(100)   # Вернет (10,20,1) -> [10:20]
s.indices(15)    # Вернет (10,15,1) -> [10:15]
```

Объект Ellipsis

Объект Ellipsis используется как признак наличия многоточия (*ellipsis*) в операторе индексирования []. Существует единственный объект данного типа, доступ к которому осуществляется с помощью встроенного имени Ellipsis. Он не имеет атрибутов и в логическом контексте оценивается, как значение True. Объект Ellipsis не используется ни в одном из встроенных типов языка Python, но он может быть полезен для тех, кто пожелает расширить функциональность оператора индексирования [] в своих собственных объектах. Ниже демонстрируется, как создается объект Ellipsis и как он передается оператору индексирования:

```
class Example(object):
    def __getitem__(self,index):
        print(index)

e = Example()
e[3, ..., 4]     # Вызывает e.__getitem__((3, Ellipsis, 4))
```

Поведение объектов и специальные методы

Вообще объекты в языке Python классифицируются по их поведению и особенностям, которые они реализуют. Например, все последовательности, такие как строки, списки и кортежи, объединены в одну категорию просто потому, что они поддерживают общий набор операций над последовательностями, таких как *s*[*n*], len(*s*) и так далее. Все основные операции интерпретатора реализованы в виде специальных методов объектов. Имена специальных методов всегда начинаются и оканчиваются двумя символами подчеркивания (__). Эти методы автоматически вызываются интерпретатором в процессе выполнения программы. Например, операция *x* + *y* отображается в вызов метода *x*.__add__(*y*), а операция доступа к элементу по индексу *x*[*k*] отображается в вызов метода *x*.__getitem__(*k*). Поведение

каждого типа данных полностью зависит от набора специальных методов, которые он реализует.

Пользовательские классы могут определять новые объекты, похожие своим поведением на встроенные типы, просто реализуя соответствующее подмножество специальных методов, о которых рассказывается в этом разделе. Кроме того, за счет переопределения некоторых специальных методов может быть расширена (через наследование) функциональность встроенных типов, таких как списки и словари.

В следующих нескольких разделах описываются специальные методы, связанные с различными категориями особенностей интерпретатора.

Создание и уничтожение объектов

Методы, перечисленные в табл. 3.11, создают, инициализируют и уничтожают экземпляры. Метод `__new__()` – это метод класса, который вызывается для создания экземпляра. Метод `__init__()` инициализирует атрибуты объекта и вызывается сразу же после создания этого объекта. Метод `__del__()` вызывается перед уничтожением объекта. Вызов этого метода происходит, только когда объект нигде больше не используется. Важно заметить, что инструкция `del x` всего лишь уменьшает счетчик ссылок на объект и необязательно может приводить к вызову этого метода. Подробнее эти методы описываются в главе 7.

Таблица 3.11. Специальные методы создания и уничтожения объектов

Метод	Описание
`__new__(cls [,*args [,**kwargs]])`	Метод класса. Вызывается для создания нового экземпляра
`__init__(self [,*args [,**kwargs]])`	Вызывается для инициализации нового экземпляра
`__del__(self)`	Вызывается перед уничтожением нового экземпляра

Методы `__new__()` и `__init__()` используются совместно для создания и инициализации новых экземпляров. Когда для создания нового экземпляра производится вызов `A(args)`, он транслируется в следующую последовательность действий:

```
x = A.__new__(A,args)
is isinstance(x,A): x.__init__(args)
```

В пользовательских объектах редко приходится определять собственные методы `__new__()` и `__del__()`. Метод `__new__()` обычно определяется только в метаклассах или в пользовательских классах, наследующих один из неизменяемых типов данных (целые числа, строки, кортежи и так далее). Метод `__del__()` определяется только в ситуациях, когда необходимо организовать управление некоторыми критически важными ресурсами, например освободить блокировку или закрыть соединение.

Строковое представление объектов

В табл. 3.12 перечислены методы, используемые для создания различных строковых представлений объектов.

Таблица 3.12. Специальные методы для создания представлений объектов

Метод	Описание
`__format__(self, format_spec)`	Создает форматированное строковое представление
`__repr__(self)`	Создает строковое представление объекта
`__str__(self)`	Создает простое строковое представление объекта

Методы `__repr__()` и `__str__()` создают простое строковое представление объекта. Метод `__repr__()` обычно возвращает строку с выражением, которое может использоваться для воссоздания объекта с помощью функции `eval()`. Этот метод отвечает также за создание выводимых значений, которые можно наблюдать при инспектировании значений в интерактивной оболочке интерпретатора. Этот метод вызывается встроенной функцией `repr()`. Ниже приводится пример совместного использования функций `repr()` и `eval()`:

```
a = [2,3,4,5] # Создание списка
s = repr(a)   # s = '[2, 3, 4, 5]'
b = eval(s)   # Строка преобразуется обратно в список
```

Если по каким-то причинам невозможно создать строковое представление выражения, позволяющего воссоздать объект, по соглашениям метод `__repr__()` должен возвращать строку вида `<...сообщение...>`, как показано ниже:

```
f = open("foo")
a = repr(f)     # a = "<open file 'foo', mode 'r' at dc030>"
```

Метод `__str__()` вызывается встроенной функцией `str()`, а также функциями, отвечающими за вывод информации. Его отличие от метода `__repr__()` заключается в том, что возвращаемая им строка должна быть более краткой и информативной для пользователя. В случае отсутствия этого метода будет вызываться метод `__repr__()`.

Метод `__format__()` вызывается обычной функцией `format()` или строковым методом `format()`. Аргумент *format_spec* – это строка, содержащая определение формата. Строка, относительно которой вызывается метод `format()`, должна иметь тот же вид, что и аргумент *format_spec* функции `format()`. Например:

```
format(x,"spec")          # Вызовет x.__format__("spec")
"x is {0:spec}".format(x) # Вызовет x.__format__("spec")
```

Синтаксис спецификаторов формата может быть какой угодно и может изменяться от объекта к объекту. Однако существует стандартный синтаксис, который описывается в главе 4.

Сравнение и упорядочение объектов

В **табл. 3.13 перечислены методы, которые могут использоваться для выполнения** простейших проверок объектов. Метод `__bool__()` используется для вычисления признака истинности и должен возвращать `True` или `False`. Если этот метод не определен, для вычисления признака истинности используется метод `__len__()`. Метод `__hash__()` определяется в объектах, которые должны предоставлять возможность использовать их в качестве ключей словаря. Он должен возвращать одно и то же целое число для объектов, которые при сравнении признаются эквивалентными. Более того, этот метод не должен определяться в изменяемых объектах; любые изменения в объекте будут приводить к изменению его хеша, что приведет к невозможности обнаружения объекта в словаре.

Таблица 3.13. Специальные методы проверки объектов и вычисления их хешей

Метод	Описание
`__bool__(self)`	Возвращает `False` или `True` при вычислении значения объекта в логическом контексте
`__hash__(self)`	Вычисляет целочисленную хеш-сумму объекта

Объекты могут реализовать поддержку одного или более операторов отношений (`<`, `>`, `<=`, `>=`, `==`, `!=`). Каждый из этих методов принимает два аргумента и может возвращать объекты любого типа, включая логический тип Boolean, список или любой другой тип языка Python. Например, пакет, реализующий численные методы, мог бы использовать эти методы для поэлементного сравнения двух матриц и возвращать матрицу с результатами. Если сравнение не может быть произведено, эти методы могут возбуждать исключение. Специальные методы, реализующие операторы сравнения, перечислены в табл. 3.14.

Таблица 3.14. Методы, реализующие поддержку операторов сравнения

Метод	Результат
`__lt__(self,other)`	`self < other`
`__le__(self,other)`	`self <= other`
`__gt__(self,other)`	`self > other`
`__ge__(self,other)`	`self >= other`
`__eq__(self,other)`	`self == other`
`__ne__(self,other)`	`self != other`

Объект не обязательно должен реализовывать все методы поддержки операторов сравнения, перечисленные в табл. 3.14. Однако если предполагается, что объект может участвовать в сравнении оператором == или использоваться в качестве ключа словаря, он должен реализовать метод __eq__(). Если предполагается, что объект может участвовать в операциях сортировки или передаваться таким функциям, как min() и max(), он должен реализовывать, по меньшей мере, метод __lt__().

Проверка типа

В табл. 3.15 перечислены методы, которые могут использоваться для переопределения поведения операций проверки типа, выполняемых функциями isinstance() b issubclass(). В большинстве случаев эти методы определяются в абстрактных базовых классах и интерфейсах, как описывается в главе 7.

Таблица 3.15. Методы проверки типа

Метод	Результат
__instancecheck__(*cls*,*object*)	isinstance(*object*, *cls*)
__subclasscheck__(*cls*, *sub*)	issubclass(*sub*, *cls*)

Доступ к атрибутам

В табл. 3.16 перечислены методы, которые используются для чтения/изменения и удаления атрибутов объектов оператором точки (.) и инструкцией del соответственно.

Таблица 3.16. Специальные методы доступа к атрибутам

Метод	Описание
__getattribute__(*self*,*name*)	Возвращает атрибут *self.name*.
__getattr__(*self*, *name*)	Возвращает атрибут *self.name*, который не может быть найден обычным способом, или возбуждает исключение AttributeError.
__setattr__(*self*, *name*, *value*)	Изменяет значение атрибута при выполнении операции *self.name = value*. Переопределяет механизм присваивания, используемый по умолчанию.
__delattr__(*self*, *name*)	Удаляет атрибут *self.name*.

Метод __getattribute__() вызывается всякий раз, когда производится обращение к атрибуту. Если методу удается обнаружить требуемый атрибут, он возвращает его значение. В противном случае вызывается метод __getattr__(). По умолчанию метод __getattr__() возбуждает исключение AttributeError. При присваивании значения атрибуту всегда вызывается метод __setattr__(), а при удалении атрибута всегда вызывается метод __delattr__().

Обертывание атрибутов и дескрипторы

Важным аспектом управления атрибутами является возможность обернуть атрибуты объекта дополнительным уровнем логики, которая выполняется при вызове операций чтения, изменения и удаления, описанных в предыдущем разделе. Этот вид обертывания обеспечивается за счет создания объекта *дескриптора*, реализующего один или более методов, перечисленных в табл. 3.17. Следует заметить, что дескрипторы не являются обязательными и необходимость в них на практике возникает достаточно редко.

Таблица 3.17. Специальные методы объектов-дескрипторов

Метод	Описание
`__get__(self,instance,cls)`	Возвращает значение атрибута или возбуждает исключение `AttributeError`
`__set__(self,instance,value)`	Записывает в атрибут значение *value*
`__delete__(self,instance)`	Удаляет атрибут

Методы `__get__()`, `__set__()` и `__delete__()` дескриптора предназначены для взаимодействия с реализацией по умолчанию методов `__getattribute__()`, `__setattr__()` и `__delattr__()` **в классах и типах. Такое взаимодействие становится возможным, если поместить экземпляр дескриптора в тело пользовательского класса. В этом случае все операции с атрибутом, определенным с помощью дескриптора, неявно будут вызывать соответствующие методы объекта дескриптора. Обычно дескрипторы используются для реализации низкоуровневой функциональности объектно-ориентированных механизмов, включая поддержку связанных и несвязанных методов, методов классов, статических методов и свойств. Примеры использования дескрипторов приводятся в главе 7.**

Методы последовательностей и отображений

В табл. 3.18 перечислены методы, используемые объектами, имитирующими поведение последовательностей и отображений.

Таблица 3.18. Методы последовательностей и отображений

Метод	Описание
`__len__(self)`	Возвращает длину объекта *self*
`__getitem__(self, key)`	Возвращает *self*[*key*]
`__setitem__(self, key, value)`	Реализует присваивание *self*[*key*] = *value*
`__delitem__(self, key)`	Удаляет *self*[*key*]
`__contains__(self, obj)`	Возвращает `True`, если *obj* присутствует в *self*; в противном случае возвращает `False`

Примеры:

```
a = [1,2,3,4,5,6]
len(a)          # a.__len__()
x = a[2]        # x = a.__getitem__(2)
a[1] = 7        # a.__setitem__(1,7)
del a[2]        # a.__delitem__(2)
5 in a          # a.__contains__(5)
```

Метод `__len__()` вызывается встроенной функцией `len()` и должен возвращать неотрицательное значение длины. Кроме того, в случае отсутствия метода `__bool__()` этот метод также используется при определении значения истинности объекта.

Для манипулирования отдельными элементами может использоваться метод `__getitem__()`, который должен возвращать элемент, соответствующий указанному ключу *key*. В качестве ключа может использоваться любой объект языка Python, но для последовательностей обычно используются целые числа. Метод `__setitem__()` **реализует операцию присваивания значения** *value* элементу. Метод `__delitem__()` вызывается всякий раз, когда вызывается инструкция `del` для отдельного элемента. Метод `__contains__()` обеспечивает реализацию оператора `in`.

Кроме того, с применением методов `__getitem__()`, `__setitem__()` и `__delitem__()` реализуются операции получения срезов, такие как `x = s[i:j]`. Однако в этом случае в качестве ключа передается специальный объект среза – `slice`. Данный объект имеет атрибуты, описывающие параметры запрошенного среза. Например:

```
a = [1,2,3,4,5,6]
x = a[1:5]          # x = a.__getitem__(slice(1,5,None))
a[1:3] = [10,11,12] # a.__setitem__(slice(1,3,None), [10,11,12])
del a[1:4]          # a.__delitem__(slice(1,4,None))
```

Операции получения среза, реализованные в языке **Python, в действительности** обладают более широкими возможностями, чем полагают многие программисты. Например, ниже приводятся поддерживаемые разновидности операции получения срезов, которые могут быть полезны при работе с многомерными структурами данных, такими как матрицы и массивы:

```
a = m[0:100:10]          # Срез с шагом (шаг=10)
b = m[1:10, 3:20]        # Многомерный срез
c = m[0:100:10, 50:75:5] # Многомерный срез с шагом
m[0:5, 5:10] = n         # Расширенная операция присваивания срезу
del m[:10, 15:]          # Расширенная операция удаления среза
```

В общем случае для каждого измерения может применяться своя операция среза в виде *i*:*j*[:*stride*], где значение *stride* (шаг) является необязательным. Как и в случае обычных срезов, допускается опускать начальное и конечное значения для каждого измерения. Кроме того, в расширенных операциях получения срезов допускается использовать многоточие (записывается как ...) для любого количества начальных и конечных измерений:

```
a = m[..., 10:20] # расширенная операция извлечения среза с многоточием
m[10:20, ...] = n
```

При использовании расширенных операций получения срезов реализация чтения, изменения и удаления элементов возлагается на методы `__getitem__()`, `__setitem__()` и `__delitem__()` соответственно. Однако вместо целочисленных значений этим методам передаются кортежи, содержащие комбинации параметров срезов и объектов `Ellipsis`. Например, инструкция

```
a = m[0:10, 0:100:5, ...]
```

вызовет метод `__getitem__()`:

```
a = m.__getitem__((slice(0,10,None), slice(0,100,5), Ellipsis))
```

Строки, кортежи и списки в языке Python в настоящее время обеспечивают некоторую поддержку расширенного синтаксиса срезов, который описывается в главе 4. Специальные расширения Python, особенно связанные с научными вычислениями, могут предлагать новые типы и объекты с расширенной поддержкой операций получения срезов.

Итерации

Если объект *obj* поддерживает итерации, он должен реализовать метод *obj*`.__iter__()`, возвращающий объект итератора. Объект итератора *iter*, в свою очередь, должен реализовать единственный метод *iter*`.next()` (или *iter*`.__next__()` в Python 3), возвращающий следующий объект или возбуждающий исключение `StopIteration` в конце итераций. Оба эти метода используются инструкцией `for` и другими инструкциями, неявно выполняющими итерации. Например, инструкция `for` *x* `in` *s* выполняет следующие действия:

```
_iter = s.__iter__()
while 1:
    try:
        x = _iter.next()    # _iter.__next__() в Python 3
    except StopIteration:
        break
    # Инструкции, составляющие тело цикла for
    ...
```

Математические операции

В табл. 3.19 перечислены специальные методы объектов, которые имитируют поведение чисел. Математические операции всегда выполняются слева направо, с учетом правил предшествования, описываемых в главе 4; когда в программе встречается выражение, такое как *x* `+` *y*, интерпретатор пытается вызвать метод *x*`.__add__(`*y*`)`. Специальные методы, имена которых начинаются с символа `r`, реализуют поддержку операций, где операнды меняются местами. Они вызываются, только если левый операнд не реализует поддержку требуемой операции. Например, если объект *x*, используемый в выражении *x* `+` *y*, не имеет метода `__add__()`, интерпретатор попытается вызвать метод *y*`.__radd__(`*x*`)`.

Таблица 3.19. Методы математических операций

Метод	Результат
`__add__(self,other)`	`self + other`
`__sub__(self,other)`	`self - other`
`__mul__(self,other)`	`self * other`
`__div__(self,other)`	`self / other` **(только в Python 2)**
`__truediv__(self,other)`	`self / other` **(Python 3)**
`__floordiv__(self,other)`	`self // other`
`__mod__(self,other)`	`self % other`
`__divmod__(self,other)`	`divmod(self,other)`
`__pow__(self,other [,modulo])`	`self ** other`, `pow(self, other, modulo)`
`__lshift__(self,other)`	`self << other`
`__rshift__(self,other)`	`self >> other`
`__and__(self,other)`	`self & other`
`__or__(self,other)`	`self | other`
`__xor__(self,other)`	`self ^ other`
`__radd__(self,other)`	`other + self`
`__rsub__(self,other)`	`other - self`
`__rmul__(self,other)`	`other * self`
`__rdiv__(self,other)`	`other / self` **(только в Python 2)**
`__rtruediv__(self,other)`	`other / self` **(Python 3)**
`__rfloordiv__(self,other)`	`other // self`
`__rmod__(self,other)`	`other % self`
`__rdivmod__(self,other)`	`divmod(other,self)`
`__rpow__(self,other)`	`other ** self`
`__rlshift__(self,other)`	`other << self`
`__rrshift__(self,other)`	`other >> self`
`__rand__(self,other)`	`other & self`
`__ror__(self,other)`	`other | self`
`__rxor__(self,other)`	`other ^ self`
`__iadd__(self,other)`	`self += other`
`__isub__(self,other)`	`self -= other`
`__imul__(self,other)`	`self *= other`
`__idiv__(self,other)`	`self /= other` **(только в Python 2)**
`__itruediv__(self,other)`	`self /= other` **(Python 3)**

Метод	Результат
`__ifloordiv__(self,other)`	*self //= other*
`__imod__(self,other)`	*self %= other*
`__ipow__(self,other)`	*self **= other*
`__iand__(self,other)`	*self &= other*
`__ior__(self,other)`	*self |= other*
`__ixor__(self,other)`	*self ^= other*
`__ilshift__(self,other)`	*self <<= other*
`__irshift__(self,other)`	*self >>= other*
`__neg__(self)`	*-self*
`__pos__(self)`	*+self*
`__abs__(self)`	`abs(self)`
`__invert__(self)`	*~self*
`__int__(self)`	`int(self)`
`__long__(self)`	`long(self)` **(только в Python 2)**
`__float__(self)`	`float(self)`
`__complex__(self)`	`complex(self)`

Методы `__iadd__()`, `__isub__()` и т. д. используются для поддержки арифметических операций, сохраняющих результат в одном из операндов, таких как `a+=b` и `a-=b` (известных также как *комбинированные операции присваивания*). Такое различие между этими операциями и стандартными арифметическими методами было введено с целью реализовать возможность попутного изменения одного из операндов и тем самым обеспечить дополнительный прирост скорости выполнения операций. Например, если объект в аргументе *self* не используется другими объектами, его можно было бы изменить непосредственно, избежав необходимости выделять память для вновь создаваемого объекта с результатом.

Существуют три разновидности операторов деления, `__div__()`, `__truediv__()` и `__floordiv__()`, которые реализуют операции истинного деления (/) и деления с усечением дробной части (//). Наличие трех разновидностей связано с изменением семантики целочисленного деления, появившейся в версии Python 2.2, которая в Python 3 была утверждена семантикой оператора /, подразумеваемой по умолчанию. В Python 2 оператор / по умолчанию отображается на вызов метода `__div__()`. Для целых чисел этот метод усекает результат до целого. В Python 3 оператор деления / отображается на вызов метода `__truediv__()` и даже для целых чисел возвращает значение с плавающей точкой. Последнее приведенное поведение может быть реализовано в Python 2 через дополнительную особенность, за счет включения в программу инструкции `from __future__ import division`.

Методы `__int__()`, `__long__()`, `__float__()` и `__complex__()` преобразуют объект, выполняя приведение к одному из четырех встроенных числовых ти-

пов. Эти методы явно вызываются при выполнении операций приведения типа, таких как `int()` или `float()`. Однако эти методы не используются в математических операциях для неявного приведения типов. Например, выражение `3 + x` породит исключение `TypeError`, даже если пользовательский объект *x* будет иметь реализацию метода `__int__()` для преобразования его в целое число.

Интерфейс вызываемых объектов

Объект может имитировать поведение функции, предоставляя метод `__call__(self [,*args [, **kwargs]])`. Если объект *x* предоставляет этот метод, к нему можно обратиться, как к функции. То есть инструкция *x*(*arg1, arg2, ...*) вызовет метод *x*.`__call__`(*self, arg1, arg2, ...*). Объекты, имитирующие поведение функции, удобно использовать для создания функторов или прокси-объектов. Ниже приводится простой пример:

```
class DistanceFrom(object):
    def __init__(self,origin):
        self.origin = origin
    def __call__(self, x):
        return abs(x - self.origin)

nums = [1, 37, 42, 101, 13, 9, -20]
nums.sort(key=DistanceFrom(10))    # Отсортирует по степени близости к числу 10
```

В этом примере создается экземпляр класса `DistanceFrom`, который имитирует поведение функции с одним аргументом. Он может использоваться вместо обычной функции – например, в вызове функции `sort()`, как в данном примере.

Протокол управления контекстом

Инструкция `with` позволяет организовать выполнение последовательности инструкций под управлением другого объекта, известного как *менеджер контекста*. В общем виде эта инструкция имеет следующий синтаксис:

```
with context [ as var]:
    инструкции
```

Ожидается, что объект *context* реализует методы, перечисленные в табл. 3.20. Метод `__enter__()` вызывается при выполнении инструкции `with`. Значение, возвращаемое этим методом, помещается в переменную, определенную с помощью необязательного спецификатора `as` *var*. Метод `__exit__()` вызывается, как только поток выполнения покидает блок инструкций, ассоциированный с инструкцией `with`. В качестве аргументов методу `__exit__()` передаются тип исключения, его значение и объект с трассировочной информацией, если в процессе выполнения инструкций в блоке было возбуждено исключение. В случае отсутствия ошибок во всех трех аргументах передается значение `None`.

Таблица 3.20. Специальные методы менеджеров контекста

Метод	Описание
`__enter__(self)`	Вызывается при входе в новый контекстный блок. Возвращаемое значение помещается в переменную, указанную в спецификаторе `as` инструкции `with`.
`__exit__(self, type, value, tb)`	Вызывается, когда поток выполнения покидает контекстный блок. Если в процессе выполнения инструкций в блоке было возбуждено исключение, в аргументах *type*, *value* и *tb* передаются тип исключения, его значение и объект с трассировочной информацией. В первую очередь инструкция `with` предназначена для упрощения управления системными ресурсами, такими как открытые файлы, сетевые соединения и блокировки. Благодаря реализации этого интерфейса объект может безопасно освобождать ресурсы после выхода потока выполнения за пределы контекста, в котором этот объект используется. Дополнительное описании приводится в главе 5 «Структура программы и управление потоком выполнения».

Функция dir() и инспектирование объектов

Функция `dir()` часто используется для инспектирования объектов. Объект может сам генерировать список имен, возвращаемый функцией `dir()`, если реализует метод `__dir__(self)`. С помощью этого метода можно скрыть внутренние особенности объекта, о которых нежелательно было бы сообщать пользователю. Однако не забывайте, что пользователь может получить полный перечень определенных имен, прибегнув к исследованию атрибута `__dict__` экземпляров и классов.

4

Операторы и выражения

В этой главе описываются встроенные операторы, выражения и порядок их вычисления в языке Python. Несмотря на то что в этой главе описываются встроенные типы, тем не менее пользовательские объекты легко могут переопределять любые из этих операторов для реализации собственного поведения.

Операции над числами

Ниже перечислены операции, которые могут применяться ко всем числовым типам:

Операция	Описание
`x + y`	Сложение
`x - y`	Вычитание
`x * y`	Умножение
`x / y`	Деление
`x // y`	Деление с усечением
`x ** y`	Возведение в степень (x^y)
`x % y`	Деление по модулю (`x mod y`)
`-x`	Унарный минус
`+x`	Унарный плюс

Оператор деления с усечением (//, также известный, как *оператор деления с округлением вниз*) отсекает дробную часть и действует как с целыми числами, так и с числами с плавающей точкой. В Python 2 оператор истинного деления (/) также усекает результат до целого, если операнды являются целыми числами. То есть выражение `7/4` даст в результате `1`, а не `1.75`. Однако в Python 3 такое поведение было изменено, и теперь оператор деления

возвращает результат с плавающей точкой. Оператор деления по модулю возвращает остаток от деления `x // y`. Например, выражение `7 % 4` вернет `3`. Для чисел с плавающей точкой оператор деления по модулю возвращает остаток с плавающей точкой от деления `x // y`, то есть результат выражения `x - (x // y) * y`. Для комплексных чисел операторы деления по модулю (%) и деления с усечением (//) считаются недопустимыми.

Ниже перечислены операторы сдвига и битовые операторы, которые могут применяться только к целым числам:

Операция	Описание
`x << y`	Сдвиг влево
`x >> y`	Сдвиг вправо
`x & y`	Битовая операция «И»
`x | y`	Битовая операция «ИЛИ»
`x ^ y`	Битовая операция «исключающее ИЛИ»
`~x`	Битовая операция «НЕ»

Битовые операторы предполагают, что целые числа имеют дополнительное двоичное представление, а знаковый бит расширяется влево до бесконечности. При работе с битовыми данными, которые должны отображаться в аппаратное представление целых чисел, следует проявлять некоторую осторожность. Это обусловлено тем, что интерпретатор Python не усекает биты и допускает возможность переполнения, то есть результат может оказаться сколь угодно большой величиной.

Кроме того, ко всем числовым типам допускается применять следующие встроенные функции:

Функция	Описание
`abs(x)`	Абсолютное значение
`divmod(x,y)`	Возвращает кортеж `(x // y, x % y)`
`pow(x,y [,modulo])`	Возвращает результат выражения `(x ** y) % modulo`
`round(x,[n])`	Округляет до ближайшего кратного 10^{-n} (только для чисел с плавающей точкой)

Функция `abs()` возвращает абсолютное значение числа. Функция `divmod()` возвращает частное и остаток от деления и может применяться к любым числам, кроме комплексных. Функция `pow()` может использоваться вместо оператора `**`, но при этом она дополнительно поддерживает возможность одновременного возведения в степень и деления по модулю (часто используется в криптографических алгоритмах). Функция `round()` округляет число с плавающей точкой `x` до ближайшего, кратного 10 в степени минус `n`. Если аргумент `n` опущен, он принимается равным `0`. Если число `x` одинаково близко к двум кратным значениям, в Python 2 округление будет выполнено

до множителя, который дальше отстоит от нуля (например, значение 0.5 будет округлено до 1.0, а значение -0.5 – до -1.0). В Python 3, при одинаковой близости к двум кратным значениям, округление будет выполнено до ближайшего четного кратного (например, значение 0.5 будет округлено до 0.0, а значение 1.5 – до 2.0). Эта особенность представляет серьезную проблему переноса математических программ на Python 3.

Следующие операторы сравнения имеют стандартную математическую интерпретацию и возвращают логическое значение True, если выражение истинно, и False, если выражение ложно:

Операция	Описание
`x < y`	Меньше чем
`x > y`	Больше чем
`x == y`	Равно
`x != y`	Не равно
`x >= y`	Больше или равно
`x <= y`	Меньше или равно

Операторы сравнения можно объединять в целые последовательности, например: `w < x < y < z`. Такая последовательность эквивалентна выражению `w < x and x < y and y < z`. Такие выражения, как `x < y > z`, также считаются допустимыми, но они будут сбивать с толку тех, кто будет читать такой программный код (важно заметить, что в этом выражении значения `x` и `z` никак не сравниваются). Когда в операции сравнения участвуют комплексные числа, возникает неопределенность, поэтому в результате будет возбуждаться исключение TypeError.

Операции над числами являются допустимыми, только если операнды принадлежат одному и тому же типу. При выполнении операций над встроенными числовыми типами производится принудительное приведение типов в соответствии со следующими правилами:

1. Если один из операндов является комплексным числом, другой операнд также преобразуется в комплексное число.
2. Если один из операндов является числом с плавающей точкой, другой операнд также преобразуется в число с плавающей точкой.
3. В противном случае оба числа должны быть целыми числами и приведение типов не выполняется.

Поведение выражения, в котором наряду с числами участвуют пользовательские объекты, зависит от реализаций этих объектов. В подобных случаях, как правило, интерпретатор не пытается выполнять неявное преобразование типов.

Операции над последовательностями

Ниже перечислены операторы, которые могут применяться к разным типам последовательностей, включая строки, списки и кортежи:

Операция	Описание
`s + r`	Конкатенация
`s * n, n * s`	Создает `n` копий последовательности `s`, где `n` – целое число
`v1,v2...,  vn = s`	Распаковывание последовательности в переменные
`s[i]`	Обращение к элементу по индексу
`s[i:j]`	Получение среза
`s[i:j:stride]`	Расширенная операция получения среза
`x in s, x not in s`	Проверка на вхождение
`for x in s:`	Итерации
`all(s)`	Возвращает `True`, если все элементы последовательности `s` оцениваются, как истинные
`any(s)`	Возвращает `True`, если хотя бы один элемент последовательности `s` оценивается, как истинный
`len(s)`	Длина последовательности
`min(s)`	Минимальный элемент в последовательности `s`
`max(s)`	Максимальный элемент в последовательности `s`
`sum(s [, initial])`	Сумма элементов последовательности с необязательным начальным значением

Оператор + объединяет две последовательности одного и того же типа. Оператор `s * n` создает `n` копий последовательности. **Однако при этом выполняется лишь поверхностное копирование**, когда копируются только ссылки на элементы. В качестве иллюстрации рассмотрим следующий пример:

```
>>> a = [3,4,5]
>>> b = [a]
>>> c = 4*b
>>> c
[[3, 4, 5], [3, 4, 5], [3, 4, 5], [3, 4, 5]]
>>> a[0] = -7
>>> c
[[-7, 4, 5], [-7, 4, 5], [-7, 4, 5], [-7, 4, 5]]
>>>
```

Обратите внимание, как изменения коснулись сразу нескольких элементов в списке `c`. В данном случае ссылки из списка `a` были скопированы в список `b`. В процессе копирования списка `b` были созданы четыре дополнительные ссылки на каждый из его элементов. Наконец, при изменении элемен-

та в списке a это изменение коснулось всех остальных «копий» данного элемента. Такое поведение оператора умножения последовательностей зачастую становится неожиданностью и не соответствует намерениям программиста. Один из способов ликвидировать эту проблему состоит в том, чтобы вручную создать копируемую последовательность, продублировав содержимое списка a. Например:

```
a = [ 3, 4, 5 ]
c = [list(a) for j in range(4)] # Функция list() создает копию списка
```

Кроме того, для копирования объектов можно также использовать модуль copy из стандартной библиотеки.

Все последовательности могут быть распакованы в последовательность переменных. Например:

```
items = [ 3, 4, 5 ]
x,y,z = items      # x = 3, y = 4, z = 5

letters = "abc"
x,y,z = letters    # x = 'a', y = 'b', z = 'c'

datetime = ((5, 19, 2008), (10, 30, "am"))
(month,day,year),(hour,minute,am_pm) = datetime
```

При распаковывании последовательностей в переменные число этих переменных должно в точности соответствовать количеству элементов в последовательности. Кроме того, структура переменных должна точно соответствовать организации последовательности. Так, в последней строке предыдущего примера последовательность распаковывается в шесть переменных, организованных в виде двух кортежей по 3 элемента в каждом, что соответствует структуре последовательности справа. Операция распаковывания в переменные может применяться к любым типам последовательностей, включая последовательности, создаваемые итераторами и генераторами.

Оператор индексирования *s*[*n*] возвращает *n*-й объект в последовательности, где *s*[0] соответствует первому объекту. Для извлечения элементов, начиная с конца последовательности, допускается использовать отрицательные индексы. Например, выражение *s*[-1] вернет последний элемент. При попытке обратиться по индексу, выходящему за пределы последовательности, будет возбуждено исключение IndexError.

Оператор среза *s*[*i*:*j*] позволяет извлекать фрагменты последовательности, состоящие из элементов, чьи индексы *k* удовлетворяют условию *i* <= *k* < *j*. Оба значения, *i* и *j*, должны быть целыми или длинными целыми числами. Если опустить начальный и/или конечный индекс, вместо них будут использованы значения начального и конечного индекса оригинальной последовательности соответственно. Допускается использовать отрицательные индексы, в этом случае они будут откладываться относительно конца последовательности. При выходе индексов *i* и/или *j* за пределы последовательности в качестве их значений будут использованы значения начального и конечного индексов оригинальной последовательности соответственно.

В операторе среза можно дополнительно указывать значение шага, *s*[*i*:*j*:*stride*], что вынудит оператор пропускать элементы оригинальной последовательности. Однако **поведение такого оператора требует дополнительных пояснений.** Если в операторе указаны величина шага *stride*, начальный индекс *i* и конечный индекс *j*, то он вернет подпоследовательность с элементами *s*[*i*], *s*[*i*+*stride*], *s*[*i*+2**stride*] и так далее, пока не будет достигнут индекс *j* (значение которого не включается в диапазон индексов отбираемых элементов). Шаг *stride* **также может быть отрицательным числом.** Если опустить индекс *i*, вместо него будет использован индекс первого элемента последовательности, если шаг задан положительным числом, и индекс последнего элемента – если шаг задан отрицательным числом. Если опустить индекс *j*, вместо него будет использован индекс последнего элемента последовательности, если шаг *stride* задан положительным числом, и индекс первого элемента – если шаг *stride* задан отрицательным числом. Например:

```
a = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

b = a[::2]        # b = [0, 2, 4, 6, 8 ]
c = a[::-2]       # c = [9, 7, 5, 3, 1 ]
d = a[0:5:2]      # d = [0,2]
e = a[5:0:-2]     # e = [5,3,1]
f = a[:5:1]       # f = [0,1,2,3,4]
g = a[:5:-1]      # g = [9,8,7,6]
h = a[5::1]       # h = [5,6,7,8,9]
i = a[5::-1]      # i = [5,4,3,2,1,0]
j = a[5:0:-1]     # j = [5,4,3,2,1]
```

Оператор *x* `in` *s* проверяет присутствие объекта *x* в последовательности *s* и возвращает значение `True` или `False`. Аналогично оператор *x* `not in` *s* проверяет отсутствие объекта *x* в последовательности *s*. Для строк операторы `in` и `not in` принимают подстроки. Например, выражение`'hello' in 'hello world'` вернет значение `True`. Важно заметить, что оператор `in` не поддерживает шаблонные символы или регулярные выражения. Если потребуется задействовать регулярные выражения, следует использовать библиотечный модуль, например `re`.

Оператор `for` *x* `in` *s* выполняет итерации по всем элементам последовательности и подробно описывается в главе 5 «Структура программы и управление потоком выполнения». Функция `len(`*s*`)` возвращает количество элементов в последовательности. Функции `min(`*s*`)` и `max(`*s*`)` возвращают минимальное и максимальное значения в последовательности соответственно; единственное ограничение состоит в том, что элементы последовательности должны поддерживать возможность упорядочения с помощью оператора `<` (например, нет смысла искать максимальное значение в списке файловых объектов). Функция `sum(`*s*`)` возвращает сумму всех элементов в последовательности *s*, но обычно может применяться только к последовательностям числовых значений. Дополнительно эта функция может принимать необязательное начальное значение. Обычно тип этого значения определяет тип результата. Например, вызов `sum(`*items*`, decimal.Decimal(0))` вернет в результате объект типа `Decimal` (подробнее о модуле `decimal` рассказывается в главе 14 «Математика»).

Строки и кортежи являются неизменяемыми объектами и не могут изменяться после их создания. Списки могут изменяться с помощью следующих операторов:

Операция	Описание
`s[i] = x`	Присваивание элементу с указанным индексом
`s[i:j] = r`	Присваивание срезу
`s[i:j:stride] = r`	Расширенная операция присваивания срезу
`del s[i]`	Удаление элемента
`del s[i:j]`	Удаление среза
`del s[i:j:stride]`	Расширенная операция удаления среза

Оператор `s[i] = x` изменяет значение элемента списка с индексом `i`, записывая в него объект `x` и увеличивая счетчик ссылок на объект `x`. Отрицательные индексы откладываются относительно конца списка, а при попытке присвоить значение элементу с индексом, выходящим за пределы последовательности, будет возбуждено исключение `IndexError`. Оператор присваивания срезу `s[i:j] = r` заменит все элементы с индексами `k`, где `i <= k < j`, на элементы из последовательности `r`. Значения индексов подчиняются тем же правилам, что и в операторе получения среза, и при выходе за допустимый диапазон принимаются равными индексам начала или конца списка. При необходимости размер последовательности `s` увеличивается или уменьшается, в зависимости от количества элементов в последовательности `r`. Например:

```
a = [1,2,3,4,5]
a[1] = 6             # a = [1,6,3,4,5]
a[2:4] = [10,11]     # a = [1,6,10,11,5]
a[3:4] = [-1,-2,-3] # a = [1,6,10,-1,-2,-3,5]
a[2:] = [0]          # a = [1,6,0]
```

Оператор присваивания срезу может принимать дополнительный параметр шага. Однако это накладывает дополнительные ограничения, в том смысле, что последовательность справа от оператора присваивания должна содержать количество элементов, в точности соответствующее количеству элементов в срезе. Например:

```
a = [1,2,3,4,5]
a[1::2] = [10,11]    # a = [1,10,3,11,5]
a[1::2] = [30,40,50] # ValueError. В срезе слева имеется всего два элемента
```

Оператор `del s[i]` удалит из списка элемент с индексом `i` и уменьшит его счетчик ссылок. Оператор `del s[i:j]` удалит из списка все элементы, попавшие в срез. Допускается указывать дополнительный параметр шага, например: `del s[i:j:stride]`.

Последовательности могут сравниваться с помощью операторов `<`, `>`, `<=`, `>=`, `==` и `!=`. В этом случае сравниваются первые элементы двух последовательностей. Их отличие определяет результат операции. Если они равны,

сравниванию подвергаются вторые элементы последовательностей. Этот процесс продолжается, пока не будут найдены различающиеся элементы или пока не будут исчерпаны все элементы любой из последовательностей. Если одновременно был достигнут конец обеих последовательностей, они признаются равными. Если *a* является частью последовательности *b*, то будет признано, что *a* < *b*.

Строки сравниваются по порядку следования символов в алфавите. Каждому символу присваивается уникальный числовой индекс, определяемый кодировкой символов (такой как ASCII или Юникод). Один символ считается меньше другого, если его индекс в алфавите меньше. Следует отметить, что предыдущие простые операторы сравнения не учитывают правила упорядочения символов, которые зависят от региональных или языковых настроек. То есть эти операторы не должны использоваться для упорядочения строк в соответствии со стандартными соглашениями, принятыми для разных языков (за дополнительной информацией обращайтесь к описаниям модулей `unicodedata` и `locale`).

Еще одно предупреждение, на сей раз связанное с типами строк. В языке Python имеется два типа строковых данных: байтовые строки и строки Юникода. Байтовые строки отличаются от строк Юникода тем, что как правило, они уже содержат символы в определенной кодировке, тогда как строки Юникода содержат некодированные значения символов. По этой причине никогда не следует смешивать байтовые строки и строки Юникода в выражениях или сравнивать их (например, использовать оператор `+` для конкатенации байтовой строки со строкой Юникода или использовать оператор `==` для сравнивания таких разнородных строк). При попытке выполнения операций над разнотипными строками в Python 3 возбуждается исключение `TypeError`, но в Python 2 в таких случаях выполняется неявное приведение байтовых строк к типу строк Юникода. Эта особенность **Python 2 во многих случаях считается ошибкой и часто является источником** непредвиденных исключений и необъяснимого поведения программы. Поэтому, чтобы избавить себя от лишней головной боли, никогда не смешивайте разнотипные строки в операциях над последовательностями.

Форматирование строк

Оператор деления по модулю (*s* `%` *d*) используется для форматирования строк, где *s* – строка формата, а *d* – коллекция объектов в виде кортежа или отображения (словаря). Поведение этого оператора напоминает функцию `sprintf()` в языке C. Строка формата может содержать элементы двух типов: обычные символы (которые остаются в неизменном виде) и спецификаторы формата, каждый из которых замещается форматированным строковым представлением элементов в кортеже или в отображении. Если аргумент *d* является кортежем, количество спецификаторов формата должно в точности соответствовать количеству объектов в кортеже *d*. Если аргумент *d* является отображением, каждый спецификатор формата должен быть ассоциирован с допустимым именем ключа в отображении (с использованием круглых скобок, как будет описано чуть ниже). Каждый спецификатор

должен начинаться с символа % и заканчиваться одним из символов форматирования, перечисленных в табл. 4.1.

Таблица 4.1. Спецификаторы формата

Символ формата	Выходной формат
d, i	Целое или длинное целое десятичное число.
u	Целое или длинное целое число без знака.
o	Целое или длинное целое восьмеричное число.
x	Целое или длинное целое шестнадцатеричное число.
X	Целое или длинное целое шестнадцатеричное число (с символами в верхнем регистре).
f	Число с плавающей точкой в формате [-]*m.dddddd*.
e	Число с плавающей точкой в формате [-]*m.dddddde±xx*.
E	Число с плавающей точкой в формате [-]*m.ddddddE±xx*.
g, G	Использует спецификатор %e или %E, если экспонента меньше -4 или больше заданной точности; в противном случае использует спецификатор %f.
s	Строка или любой объект. Для создания строкового представления объектов используется функция str().
r	Воспроизводит ту же строку, что и функция repr().
c	Единственный символ.
%	Символ %.

Между символом % и символом формата могут добавляться следующие модификаторы в указанном порядке:

1. Имя ключа в круглых скобках, с помощью которого выбирается элемент из объекта отображения. Если такой элемент отсутствует, возбуждается исключение KeyError.
2. Один или более следующих символов:
   - Знак -, определяет выравнивание по левому краю. По умолчанию значения выравниваются по правому краю.
   - Знак +, указывает на необходимость обязательного включения знака для числовых значений (даже для положительных).
   - 0, указывает на необходимость заполнения нулями.
3. Число, определяющее минимальную ширину поля. Форматируемое значение будет выводиться в поле, ширина которого не может быть меньше указанной, и дополняться добавочными символами слева (или справа, если указан флаг -) до указанной ширины поля.
4. Точка, отделяющая значение ширины поля от значения точности.
5. Число, определяющее максимальное количество символов при выводе строк; количество цифр после десятичной точки, при выводе чисел

с плавающей точкой, или минимальное количество цифр, при выводе целых чисел.

Кроме того, в модификаторах вместо числовых значений ширины поля и точности допускается использовать символ звездочки (*). В этом случае значения ширины и точности будут браться из элементов кортежа, предшествующих форматируемому значению.

В следующем фрагменте приводится несколько примеров:

```
a = 42
b = 13.142783
c = "hello"
d = {'x':13, 'y':1.54321, 'z':'world'}
e = 5628398123741234
r = "a is %d" % a             # r = "a is 42"
r = "%10d %f" % (a,b)         # r = "  42 13.142783"
r = "%+010d %E" % (a,b)       # r = "+000000042 1.314278E+01"
r = "%(x)-10d %(y)0.3g" % d   # r = "13 1.54"
r = "%0.4s %s" % (c, d['z']) # r = "hell world"
r = "%*.*f" % (5,3,b)         # r = "13.143"
r = "e = %d" % e              # r = "e = 5628398123741234"
```

При использовании со словарями оператор форматирования строк % часто используется для имитации такой особенности, как подстановка значений переменных в строки, которую нередко можно встретить в языках сценариев (такой как подстановка значения переменной `$var` в строку). Например, если имеется некоторый словарь, значения его элементов можно вставить в строку формата, как показано ниже:

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

r = "%(shares)d акций компании %(name)s по цене %(price)0.2f" % stock
# r = "100 акций компании GOOG по цене 490.10"
```

Следующий фрагмент демонстрирует, как можно вставить в строку текущие значения переменных. Функция `vars()` возвращает словарь, содержащий все переменные, определенные к моменту ее вызова.

```
name = "Элвуд"
age = 41
r = "%(name)s, возраст %(age)s год" % vars()
```

Дополнительные возможности форматирования

Строковый метод *s*.`format`(**args*, ***kwargs*) обеспечивает дополнительные возможности форматирования. Этот метод принимает произвольные коллекции позиционных и именованных аргументов и подставляет их значения на место символов подстановки в строке *s*. Символы подстановки в формате '{*n*}', где *n* – число, замещаются соответствующими позиционными аргументами метода `format()`. Символы подстановки в формате '{*name*}'

замещаются соответствующими именованными аргументами. Чтобы вывести один символ '{', следует использовать '{{', и '}}' – чтобы вывести один символ '}'. Например:

```
r = "{0} {1} {2}".format('GOOG',100,490.10)
r = "{name} {shares} {price}".format(name='GOOG',shares=100,price=490.10)
r = "Привет, {0}, ваш возраст {age} год".format("Элвуд",age=47)
r = "Комбинации {{ и }} используются для вывода фигурных скобок".format()
```

Каждый символ подстановки может сопровождаться дополнительным индексом или именем атрибута. Например, в строке '{*name*[*n*]}', где *n* – целое число, на место символа подстановки будет вставлен *n*-й элемент последовательности *name*, а в строке '{*name*[*key*]}', где *key* – нечисловая строка, будет вставлен элемент словаря *name*['*key*']. В строке '{*name.attr*}' на место символа подстановки будет вставлено значение атрибута *name.attr*. Например:

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

r = "{0[name]} {0[shares]} {0[price]}".format(stock)
x = 3 + 4j
r = "{0.real} {0.imag}".format(x)
```

В этих символах подстановки допускается использовать только действительные имена. Произвольные выражения, вызовы методов и другие операции не поддерживаются.

Для более точного управления результатами вместе с символами подстановки можно использовать спецификаторы формата. Для этого к каждому символу подстановки можно добавить необязательный спецификатор формата, через двоеточие (:), например: '{*place*:*format_spec*}'. При таком подходе можно указывать ширину поля, количество десятичных разрядов и выравнивание. Ниже приводятся несколько примеров:

```
r = "{name:8} {shares:8d} {price:8.2f}".format
(name="GOOG",shares=100,price=490.10)
```

В общем виде спецификатор формата имеет вид [[*fill*[*align*]][*sign*][0][*width*][.*precision*][*type*], где каждая часть, заключенная в [], является необязательной. Значение *width* определяет минимальную ширину поля, а в качестве спецификатора *align* можно использовать символ '<', '>' или '^', для выравнивания по левому краю, по правому краю или по центру. Необязательное значение *fill* определяет символ, который будет использоваться для дополнения результата до требуемой ширины поля. Например:

```
name = "Elwood"
r = "{0:<10}".format(name) # r = 'Elwood    '
r = "{0:>10}".format(name) # r = '    Elwood'
r = "{0:^10}".format(name) # r = '  Elwood  '
r = "{0:=^10}".format(name) # r = '==Elwood=='
```

Значение *type* определяет тип данных. В табл. 4.2 перечислены все поддерживаемые типы форматируемых данных. Если значение *type* не указано,

по умолчанию для строк используется код 's', для целых чисел – 'd' и для чисел с плавающей точкой – 'f'.

Таблица 4.2. Дополнительные коды типов для спецификаторов формата

Символ формата	Выходной формат
d, i	Целое или длинное целое десятичное число.
b	Целое или длинное целое двоичное число.
o	Целое или длинное целое восьмеричное число.
x	Целое или длинное целое шестнадцатеричное число.
X	Целое или длинное целое шестнадцатеричное число (с символами в верхнем регистре).
f, F	Число с плавающей точкой в формате [-]*m.dddddd*.
e	Число с плавающей точкой в формате [-]*m.dddddde±xx*.
E	Число с плавающей точкой в формате [-]*m.ddddddE±xx*.
g, G	Использует спецификатор e или E, если экспонента меньше -4 или больше заданной точности; в противном случае использует спецификатор f.
n	То же, что и g, только символ десятичной точки определяется в соответствии с региональными настройками.
%	Умножает число на 100 и отображает его с использованием формата f, завершая символом %.
s	Строка или любой другой объект. Для создания строкового представления объекта используется функция str().
c	Одиночный символ.

В качестве значения *sign* в спецификаторе формата допускается использовать символ '+', '-' или ' '. Значение '+' указывает, что перед всеми числами должен добавляться их знак. Значение '-' указывает, что знак должен добавляться только для отрицательных чисел. Если указано значение ' ', перед положительными числами добавляется ведущий пробел. Значение *precision* определяет количество цифр после запятой для десятичных чисел. Если перед значением *width* ширины поля указан ведущий символ '0', числовые значения будут дополняться слева ведущими нулями до требуемой ширины поля. Ниже приводятся некоторые примеры форматирования различных числовых значений:

```
x = 42
r = '{0:10d}'.format(x)      # r = '        42'
r = '{0:10x}'.format(x)      # r = '        2a'
r = '{0:10b}'.format(x)      # r = '    101010'
r = '{0:010b}'.format(x)     # r = '0000101010'
y = 3.1415926
r = '{0:10.2f}'.format(y)    # r = '      3.14'
r = '{0:10.2e}'.format(y)    # r = '  3.14e+00'
```

```
r = '{0:+10.2f}'.format(y)   # r = '  +3.14'
r = '{0:+010.2f}'.format(y)  # r = '+000003.14'
r = '{0:+10.2%}'.format(y)   # r = '  +314.16%'
```

При необходимости значения различных элементов спецификаторов формата могут определяться в виде аргументов, передаваемых методу `format()`. Доступ к ним осуществляется как и к другим элементам строки формата. Например:

```
y = 3.1415926
r = '{0:{width}.{precision}f}'.format(y,width=10,precision=3)
r = '{0:{1}.{2}f}'.format(y,10,3)
```

При таком подходе подставляемые значения могут определяться только на первом уровне вложенности и должны соответствовать элементам спецификаторов формата. Кроме того, у подставляемых значений не может быть собственных спецификаторов формата.

Следует заметить, что объекты могут определять собственный набор спецификаторов формата. За кулисами метод `format()` вызывает специальный метод `__format__(`*self*, *format_spec*`)` для каждого форматируемого значения. Благодаря этому имеется возможность изменить поведение метода `format()` и реализовать обработку спецификаторов в объекте. Например, даты, время и объекты других типов могут определять свои собственные коды форматирования.

В некоторых случаях может потребоваться просто отформатировать строковое представление объекта, воспроизводимое функциями `str()` и `repr()`, в обход функциональности, реализованной в методе `__format__()`. Для этого перед спецификатором формата достаточно добавить модификатор `'!s'` или `'!r'`. Например:

```
name = "Гвидо"
r = '{0!r:^20}'.format(name)    # r = "  'Гвидо'  "
```

Операции над словарями

Словари реализуют отображение между именами и объектами. К ним могут применяться следующие операции:

Операция	Описание
x = *d*[*k*]	Обращение к элементу с ключом *key*
d[*k*] = *x*	Присваивание элементу с ключом *key*
del *d*[*k*]	Удаление элемента с ключом *key*
k in *d*	Проверка наличия элемента с ключом *ke*y
len(*d*)	Количество элементов в словаре

В качестве ключей допускается использовать любые неизменяемые объекты, такие как строки, числа и кортежи. Кроме того, в качестве ключей

можно использовать списки значений, разделенные запятыми, как показано ниже:

```
d = { }
d[1,2,3] = "foo"
d[1,0,3] = "bar"
```

В данном случае в качестве ключей используются кортежи. Учитывая эту особенность, предыдущие операции присваивания можно записать так:

```
d[(1,2,3)] = "foo"
d[(1,0,3)] = "bar"
```

Операции над множествами

Типы `set` и `frozenset` поддерживают ряд общих операций:

Операция	Описание
s | *t*	Объединение множеств *s* и *t*
s & *t*	Пересечение множеств *s* и *t*
s - *t*	Разность множеств *s* и *t*
s ^ *t*	Симметричная разность множеств *s* и *t*
len(*s*)	Количество элементов в множестве
max(*s*)	Максимальное значение
min(*s*)	Минимальное значение

Тип результата операций объединения, пересечения и разности определяется типом левого операнда. Например, если операнд *s* имеет тип `frozenset`, результат также будет иметь тип `frozenset`, даже если операнд *t* имеет тип `set`.

Комбинированные операторы присваивания

В языке Python поддерживаются следующие комбинированные операторы присваивания:

Операция	Описание
x += *y*	*x* = *x* + *y*
x -= *y*	*x* = *x* - *y*
x *= *y*	*x* = *x* * *y*
x /= *y*	*x* = *x* / *y*
x //= *y*	*x* = *x* // *y*
x **= *y*	*x* = *x* ** *y*
x %= *y*	*x* = *x* % *y*

(продолжение)

Операция	Описание
`x &= y`	`x = x & y`
`x |= y`	`x = x | y`
`x ^= y`	`x = x ^ y`
`x >>= y`	`x = x >> y`
`x <<= y`	`x = x << y`

Эти операторы можно использовать везде, где допускается использовать обычный оператор присваивания. Например:

```
a = 3
b = [1,2]
c = "Hello %s %s"
a += 1                    # a = 4
b[1] += 10                # b = [1, 12]
c %= ("Monty", "Python") # c = "Hello Monty Python"
```

Комбинированные операторы присваивания не нарушают принцип неизменяемости и не производят изменение самих объектов. То есть инструкция `x += y` создаст совершенно новый объект `x` со значением `x + y`. Пользовательские объекты могут переопределять комбинированные операторы присваивания с помощью специальных методов, описанных в главе 3 «Типы данных и объекты».

Оператор доступа к атрибутам (.)

Для доступа к атрибутам объектов используется оператор «точка» (.). Например:

```
foo.x = 3
print foo.y
a = foo.bar(3,4,5)
```

В одном выражении может использоваться сразу несколько операторов точки, например: `foo.y.a.b`. Оператор точки может также применяться к промежуточным результатам функций, например: `a = foo.bar(3,4,5).spam`.

Пользовательские объекты могут переопределять или изменять поведение оператора (.). Подробности приводятся в главе 3 и в главе 7 «Классы и объектно-ориентированное программирование».

Оператор вызова функции ()

Оператор `f(args)` используется для вызова функции `f`. Все аргументы функций представлены выражениями. Перед вызовом функции сначала вычисляются все выражения аргументов, в направлении слева направо. Иногда такой порядок вычислений называют *аппликативным*.

Имеется возможность вычислить ряд аргументов функции заранее, с помощью функции `partial()` из модуля `functools`. Например:

```
def foo(x,y,z):
    return x + y + z

from functools import partial
f = partial(foo,1,2) # Запоминает значения для аргументов x и y функции foo
f(3)                 # Вызов foo(1,2,3), возвратит число 6
```

Функция `partial()` вычисляет некоторые из аргументов указанной функции и возвращает объект, который позднее можно вызвать, передав ему недостающие аргументы. В предыдущем примере переменная `f` представляет функцию, для которой уже были вычислены значения двух первых аргументов. Для вызова этой функции достаточно просто передать ей недостающий последний аргумент. Прием предварительного вычисления значений аргументов функции тесно связан с *каррингом* (*currying*) – механизмом, с помощью которого функция, принимающая несколько аргументов, такая как `f(x,y)`, раскладывается в серию функций, каждая из которых принимает единственный аргумент. Например, зафиксировав значение аргумента `x`, для функции `f` можно создать новую, частично подготовленную функцию, которой достаточно будет передать значение `y`, чтобы получить результат.

Функции преобразования

Иногда бывает необходимо выполнить преобразование между встроенными типами. Для этого достаточно просто обратиться к имени требуемого типа, как к функции. Кроме того, существует несколько встроенных функций, выполняющих специальные виды преобразований. Все эти функции возвращают новый объект, представляющий преобразованное значение.

Функция	Описание
`int(`*x* `[,`*base*`])`	Преобразует объект *x* в целое число. Если *x* является строкой, аргумент *base* определяет основание системы счисления.
`float(`*x*`)`	Преобразует объект *x* в число с плавающей точкой.
`complex(`*real* `[,`*imag*`])`	Преобразует объект *x* в комплексное число.
`str(`*x*`)`	Преобразует объект *x* ***в строковое представление.***
`repr(`*x*`)`	Преобразует объект *x* в строковое выражение.
`format(`*x* `[,`*format_spec*`])`	Преобразует объект *x* в форматированную строку.
`eval(`*str*`)`	Вычисляет выражение в строке *str* и возвращает объект.
`tuple(s)`	Преобразует объект *s* в кортеж.
`list(s)`	Преобразует объект *s* в список.
`set(s)`	Преобразует объект *s* в множество.

(продолжение)

Функция	Описание
`dict(d)`	Создает словарь. Объект *d должен быть последовательностью кортежей* (*key, value*).
`frozenset(s)`	Преобразует объект *s* в множество типа `frozenset`.
`chr(x)`	Преобразует целое число *x* в символ.
`unichr(x)`	Преобразует целое число *x* в символ Юникода (только в Python 2).
`ord(x)`	Преобразует одиночный символ в целое число.
`hex(x)`	Преобразует целое число *x* в строку с шестнадцатеричным представлением.
`bin(x)`	Преобразует целое число *x* в строку с двоичным представлением.
`oct(x)`	Преобразует целое число *x* в строку с восьмеричным представлением.

Обратите внимание, что функции `str()` и `repr()` могут возвращать различные результаты. Функция `repr()` обычно создает строку с выражением, которое можно передать функции `eval()`, чтобы воссоздать объект, тогда как функция `str()` возвращает более краткое или отформатированное строковое представление объекта (и используется инструкцией `print`). Функция `format(x, [format_spec])` возвращает такое же строковое представление объекта, что и оператор форматирования, но применяется к единственному объекту *x*. На входе она принимает необязательный аргумент *format_spec*, который является строкой, содержащей спецификаторы формата. Функция `ord()` возвращает порядковый номер символа. Для символов Юникода это значение совпадает с целочисленным значением кодового пункта. Функции `chr()` и `unichr()` преобразуют целое число в символ.

Преобразовать строку в число можно с помощью функций `int()`, `float()` и `complex()`. Функция `eval()` также может преобразовать строку, содержащую допустимое выражение, в объект. Например:

```
a = int("34")             # a = 34
b = long("0xfe76214", 16) # b = 266822164L (0xfe76214L)
b = float("3.1415926")    # b = 3.1415926
c = eval("3, 5, 6")       # c = (3,5,6)
```

В функции, создающие контейнеры (`list()`, `tuple()`, `set()` и другие), в качестве аргумента можно передавать любые объекты, поддерживающие итерации, которые будут выполнены для создания всех элементов создаваемой последовательности.

Логические выражения и значения истинности

Логические выражения формируются с помощью ключевых слов `and`, `or` и `not`. Эти операторы отличаются следующим поведением:

Оператор	Описание
`x or y`	Если *x* ложно, возвращается значение *y*, в противном случае – значение *x*.
`x and y`	Если *x* ложно, возвращается значение *x*, в противном случае – значение *y*.
`not x`	Если *x* ложно, возвращается *1*, в противном случае – *0*.

Когда в логическом контексте определяется истинность или ложность, значение `True` принимают любые числа, отличные от нуля, а также непустые строки, списки, кортежи и словари. Значение `False` принимают число `0`, `None`, а также пустые строки, списки, кортежи и словари. Значение логического выражения вычисляется слева направо, при этом правый операнд рассматривается, только если его значение может повлиять на значение всего выражения. Например, в выражении `a and b` операнд `b` рассматривается, только если операнд `a` имеет истинное значение. Иногда такой порядок вычислений называют «*сокращенной схемой вычисления*».

Равенство и идентичность объектов

Оператор равенства (*x* `==` *y*) проверяет равенство значений *x* и *y*. В случае списков и кортежей сравниваются все элементы, и значение `True` возвращается, только если все значения равны. При сравнивании словарей значение `True` возвращается, только если словари *x* и *y* имеют одинаковый набор ключей и объекты с одним и тем же ключом имеют одинаковые значения. Два множества считаются равными, если они содержат один и тот же набор элементов, которые сравниваются между собой с помощью оператора равенства (`==`).

Операторы идентичности (*x* `is` *y* и *x* `is not` *y*) проверяют, ссылаются ли сравниваемые переменные на один и тот же объект. Вообще нет ничего необычного, когда *x* `==` *y*, но *x* `is not` *y*.

Сравнивание объектов несовместимых типов, таких как файлы и числа с плавающей точкой, может быть допустимо, но результат такого сравнивания непредсказуем и может вообще не иметь никакого смысла. Кроме того, в зависимости от типов операндов, такое сравнивание может привести к исключению.

Порядок вычисления

В табл. 4.3 приводится порядок вычисления операторов языка Python (правила предшествования). Все операторы, за исключением (`**`), вычисляются слева направо и перечислены в таблице в порядке убывания приоритета. То есть операторы, находящиеся ближе к началу таблицы, вычисляются раньше операторов, находящихся ниже. (Обратите внимание, что операторы, объединенные в один подраздел, такие как `x * y`, `x / y`, `x // y` и `x % y`, имеют одинаковый приоритет.)

Таблица 4.3. Порядок вычисления операторов (по убыванию приоритета)

Оператор	Название
`(...)`, `[...]`, `{...}`	Создание кортежа, списка и словаря
`s[i]`, `s[i:j]`	Операции доступа к элементам и срезам
`s.attr`	Операции доступа к атрибутам
`f(...)`	Вызовы функций
`+x`, `-x`, `~x`	Унарные операторы
`x ** y`	Возведение в степень (правоассоциативная операция)
`x * y`, `x / y`, `x // y`, `x % y`	Умножение, деление, деление с усечением, деление по модулю
`x + y`, `x - y`	Сложение, вычитание
`x << y`, `x >> y`	Битовый сдвиг
`x & y`	Битовая операция «И»
`x ^ y`	Битовая операция «исключающее ИЛИ»
`x | y`	Битовая операция «ИЛИ»
`x < y`, `x <= y`, `x > y`, `x >= y`, `x == y`, `x != y`, `x is y`, `x is not y`, `x in s`, `x not in s`	Сравнивание, проверка на идентичность, проверка на вхождение в состав последовательности
`not x`	Логическое отрицание
`x and y`	Логическая операция «И»
`x or y`	Логическая операция «ИЛИ»
`lambda args: expr`	Анонимная функция

Порядок вычисления операторов, перечисленных в табл. 4.3, не зависит от типов аргументов *x* и *y*. Поэтому, хотя пользовательский объект может переопределить отдельные операторы, тем не менее он не в состоянии изменить порядок их вычисления, приоритеты и правила ассоциативности.

Условные выражения

В программировании широко распространен прием присваивания того или иного значения, в зависимости от некоторого условия. Например:

```
if a <= b:
    minvalue = a
else:
    minvalue = b
```

Этот программный код можно записать короче, если воспользоваться *условным выражением*. Например:

```
minvalue = a if a <=b else b
```

Первой вычисляется условная часть, находящаяся в середине таких выражений. Если в результате получается значение `True`, вычисляется выражение *a*, находящееся левее инструкции `if`. В противном случае вычисляется выражение *b*, находящееся правее инструкции `else`.

Не следует злоупотреблять условными выражениями, потому что чрезмерное их использование может приводить к путанице (особенно в случае вложения их друг в друга или при включении их в другие сложные выражения). Однако их очень удобно использовать в генераторах списков и в выражениях-генераторах. Например:

```
values = [1, 100, 45, 23, 73, 37, 69 ]
clamped = [x if x < 50 else 50 for x in values]
print(clamped)      # [1, 50, 45, 23, 50, 37, 50]
```

5

Структура программы и управление потоком выполнения

В этой главе подробно рассматриваются структура программы и способы управления потоком выполнения. В число рассматриваемых тем входят: условные операторы, итерации, исключения и менеджеры контекста.

Структура программы и ее выполнение

Программы на языке Python **представляют собой последовательности инструкций**. Все функциональные возможности языка, включая присваивание значений переменным, определение функций и классов, импортирование модулей, реализованы в виде инструкций, обладающих равным положением со всеми остальными инструкциями. В действительности в языке Python нет «специальных» инструкций и всякая инструкция может быть помещена в любом месте в программе. Например, в следующем фрагменте определяются две различные версии функции:

```
if debug:
    def square(x):
        if not isinstance(x,float):
            raise TypeError("Expected a float")
        return x * x
else:
    def square(x):
        return x * x
```

Загружая исходные файлы программы, интерпретатор всегда последовательно выполняет все инструкции, пока не встретит конец файла. Эта модель выполнения в равной степени относится как к главному файлу программы, так и к библиотечным файлам модулей, загружаемым с помощью инструкции `import`.

Выполнение по условию

Инструкции `if`, `else` и `elif` позволяют организовать выполнение программного кода в зависимости от условий. В общем случае условная инструкция имеет следующий вид:

```
if выражение:
    инструкции
elif выражение:
    инструкции
elif выражение:
    инструкции
...
else:
    инструкции
```

Если при несоблюдении условия никаких действий не должно выполняться, можно опустить части `else` и `elif` условной инструкции. Если действия не должны выполняться при соблюдении определенного условия, можно использовать инструкцию `pass`:

```
if выражение:
    pass     # Ничего не делает
else:
    инструкции
```

Циклы и итерации

Циклы в языке Python оформляются с помощью инструкций `for` и `while`. Например:

```
while выражение:
    инструкции

for i in s:
    инструкции
```

Инструкция `while` выполняет инструкции, пока условное выражение не вернет значение `False`. Инструкция `for` выполняет итерации по всем элементам `s`, пока не исчерпаются доступные элементы. Инструкция `for` может работать с любыми объектами, поддерживающими итерации. К числу этих объектов относятся встроенные типы последовательностей, такие как списки, кортежи и строки, а также любые другие объекты, реализующие протокол итераторов.

Считается, что объект `s` поддерживает итерации, если он может использоваться в следующем программном коде, который является отражением реализации инструкции `for`:

```
it = s.__iter__()         # Получить итератор для объекта s
while 1:
    try:
        i = it.next()     # Получить следующий элемент (__next__ в Python 3)
    except StopIteration: # Нет больше элементов
```

```
            break
            # Выполнить операции над i
    ...
```

Переменная *i* в инструкции `for i in s` называется *переменной цикла*. На каждой итерации она принимает новое значение из объекта *s*. Область видимости переменной цикла не ограничивается инструкцией `for`. Если перед инструкцией `for` была объявлена переменная с тем же именем, ее предыдущее значение будет затерто. Более того, по окончании цикла эта переменная будет хранить последнее значение, полученное в цикле.

Если элементы, извлекаемые в процессе итерации, являются последовательностями одного и того же размера, их можно распаковать в несколько переменных цикла, используя прием, демонстрируемый ниже:

```
for x,y,z in s:
    инструкции
```

В этом примере объект *s* должен содержать или воспроизводить последовательности, каждая из которых содержит по три элемента. В каждой итерации переменным *x*, *y* и *z* будут присваиваться значения элементов соответствующей последовательности. Несмотря на то что наиболее часто этот прием используется, когда объект *s* является последовательностью кортежей, тем не менее он может применяться к элементам объекта *s*, являющимся последовательностями любого типа, включая списки, генераторы и строки.

При выполнении итераций, помимо самих данных, иногда бывает удобно иметь доступ к порядковому номеру итерации, как показано ниже:

```
i = 0
for x in s:
    инструкции
    i += 1
```

Для этих целей в языке Python может использоваться встроенная функция `enumerate()`, например:

```
for i,x in enumerate(s):
    инструкции
```

Вызов `enumerate(`*s*`)` создает итератор, который просто возвращает последовательность кортежей `(0, `*s*`[0])`, `(1, `*s*`[1])`, `(2, `*s*`[2])` и так далее.

Другая типичная проблема, связанная с организацией циклов, – параллельная обработка двух или более последовательностей. Например, цикл, в каждой итерации которого требуется одновременно извлекать элементы из разных последовательностей, может быть реализован так:

```
# s и t - это две последовательности
i = 0
while i < len(s) and i < len(t):
    x = s[i]     # Извлечь элемент из последовательности s
    y = t[i]     # Извлечь элемент из последовательности t
    инструкции
    i += 1
```

Эту реализацию можно упростить, если воспользоваться функцией `zip()`. Например:

```
# s и t - это две последовательности
for x,y in zip(s,t):
    инструкции
```

Функция `zip(`*s*`,`*t*`)` объединит последовательности *s* и *t* в последовательность кортежей (*s*`[0]`,*t*`[0]`), (*s*`[1]`,*t*`[1]`), (*s*`[2]`, *t*`[2]`) и так далее, остановившись после исчерпания элементов в самой короткой из последовательностей, если их длины не равны. Следует особо отметить, что в версии Python 2 функция `zip()` извлекает все элементы из последовательностей *s* и *t* и создает список кортежей. Для генераторов и последовательностей, содержащих огромные количества элементов, такое поведение может оказаться нежелательным. Эту проблему можно решить с помощью функции `itertools.izip()`, которая достигает того же эффекта, что и функция `zip()`, но вместо того чтобы создавать огромный список кортежей, она воспроизводит по одному комплекту объединенных значений при каждом обращении. В Python 3 функция `zip()` реализована точно так же.

Прервать цикл можно с помощью инструкции `break`. Например, в следующем фрагменте выполняется чтение строк из текстового файла, пока не будет встречена пустая строка:

```
for line in open("foo.txt"):
    stripped = line.strip()
    if not stripped:
        break              # Пустая строка, прервать чтение
    # Обработать строку stripped
    ...
```

Перейти к следующей итерации (пропустив оставшиеся инструкции в теле цикла) можно с помощью инструкции `continue`. Эта инструкция используется не часто, но иногда она может быть полезна, – когда оформление еще одного уровня вложенности программного кода в условной инструкции могло бы привести к нежелательному усложнению программы. Например, следующий цикл пропускает все пустые строки в файле:

```
for line in open("foo.txt"):
    stripped = line.strip()
    if not stripped:
        continue           # Пропустить пустую строку
    # Обработать строку stripped
    ...
```

Инструкции `break` и `continue` воздействуют только на самый внутренний цикл. Если необходимо организовать выход из глубоко вложенной иерархии циклов, можно воспользоваться исключением. В языке Python отсутствует инструкция «`goto`».

К инструкции цикла можно также присоединить инструкцию `else`, как показано в следующем примере:

```
# for-else
for line in open("foo.txt"):
```

```
        stripped = line.strip()
        if not stripped:
            break
        # Обработать строку stripped
        ...
    else:
        raise RuntimeError("Отсутствует разделитель параграфов")
```

Блок `else` в инструкции цикла выполняется только в случае нормального завершения цикла, то есть либо когда не было выполнено ни одной итерации, либо после выполнения последней итерации. Если выполнение цикла прерывается преждевременно с помощью инструкции `break`, блок `else` пропускается.

Основное назначение инструкции `else` в циклах заключается в том, чтобы избежать установки или проверки какого-либо флага или условия, когда работа цикла прерывается преждевременно. Например, предыдущий пример можно переписать без использования блока `else`, но с применением дополнительной переменной:

```
found_separator = False
    for line in open("foo.txt"):
        stripped = line.strip()
        if not stripped:
            found_separator = True
            break
        # Обработать строку stripped
        ...

if not found_separator:
    raise RuntimeError("Отсутствует разделитель параграфов ")
```

Исключения

Исключения свидетельствуют об ошибках и прерывают нормальный ход выполнения программы. Исключения возбуждаются с помощью инструкции `raise`. В общем случае инструкция `raise` имеет следующий вид: `raise` *`Exception`*`([`*`value`*`])`, где *`Exception`* – тип исключения, а *`value`* – необязательное значение с дополнительной информацией об исключении. Например:

```
raise RuntimeError("Неустранимая ошибка")
```

Если инструкция `raise` используется без дополнительных параметров, она повторно возбуждает последнее исключение (однако такой прием работает только в процессе обработки возникшего исключения).

Перехватить исключение можно с помощью инструкций `try` и `except`, как показано ниже:

```
try:
    f = open('foo')
except IOError as e:
    инструкции
```

Когда возникает исключение, интерпретатор прекращает выполнение инструкций в блоке `try` и отыскивает блок `except`, соответствующий типу возникшего исключения. В случае успеха управление передается первой инструкции в найденном блоке `except`. После выполнения блока `except` выполнение программы продолжается с первой инструкции, следующей за блоком `try-except`. В противном случае исключение передается блоку программного кода, вмещающему инструкцию `try`. Этот программный код в свою очередь также может быть заключен в блок `try-except`, предусматривающий обработку исключения. Если исключение достигнет самого верхнего уровня программы и не будет перехвачено, интерпретатор прервет выполнение программы с сообщением об ошибке. При необходимости все неперехваченные исключения можно обработать в пользовательской функции `sys.excepthook()`, как описывается в главе 13 «Службы Python времени выполнения».

Необязательный модификатор `as` *`var`* в инструкции `except` определяет имя переменной, в которую будет записан тип возникшего исключения. Обработчики исключений могут использовать это значение для получения более подробной информации об исключении. Например, проверку типа возникшего исключения можно выполнить с помощью инструкции `isinstance()`. Следует отметить, что в предыдущих версиях Python инструкция `except` записывалась как `except` *`ExcType`*`,` *`var`*, где тип исключения и имя переменной отделялись запятой (`,`). Этот синтаксис можно (но не рекомендуется) использовать в версии **Python 2.6. В новых программах следует использовать синтаксис** `as` *`var`*, потому что он является единственно допустимым в Python 3.

Допускается использовать несколько блоков `except` для обработки разных типов исключений, как показано в следующем примере:

```
try:
    выполнить некоторые действия
except IOError as e:
    # Обработка ошибки ввода-вывода
    ...
except TypeError as e:
    # Обработка ошибки типа объекта
    ...
except NameError as e:
    # Обработка ошибки обращения к несуществующему имени
    ...
```

Несколько типов исключений можно также обрабатывать с помощью единственного обработчика, например:

```
try:
    выполнить некоторые действия
except (IOError, TypeError, NameError) as e:
    # Обработать ошибку ввода-вывода, типа
    # или обращения к несуществующему имени
    ...
```

Игнорировать исключение можно с помощью инструкции `pass`, как показано ниже:

```
try:
    выполнить некоторые действия
except IOError:
    pass        # Ничего не делать (сойдет и так).
```

Перехватить все исключения, кроме тех, что приводят к немедленному завершению программы, можно, указав тип исключения `Exception`, как показано ниже:

```
try:
    выполнить некоторые действия
except Exception as e:
    error_log.write('Возникла ошибка: %s\n' % e)
```

Когда перехватываются все типы исключений, необходимо позаботиться о записи в журнал точной информации об ошибке для последующего анализа. Например, в предыдущем фрагменте в журнал записывается информация, имеющая отношение к исключению. Если пренебречь этой информацией, будет очень сложно отладить программу, в которой возникают ошибки, появления которых вы не ожидаете.

С помощью инструкции `except`, без указания типа исключения, можно перехватывать исключения любых типов:

```
try:
    выполнить некоторые действия
except:
    error_log.write('Ошибка\n')
```

Правильное использование этой формы инструкции `except` является намного более сложным делом, чем может показаться, поэтому лучше избегать такого способа обработки исключений. Например, этот фрагмент будет также перехватывать прерывания от клавиатуры и запросы на завершение программы, которые едва ли имеет смысл перехватывать.

Кроме того, инструкция `try` может сопровождаться блоком `else`, который должно следовать за последним блоком `except`. Этот блок выполняется, если в теле инструкции `try` не возникло исключений. Например:

```
try:
    f = open('foo', 'r')
except IOError as e:
    error_log.write('Невозможно открыть файл foo: %s\n' % e)
else:
    data = f.read()
    f.close()
```

Инструкция `finally` служит для реализации завершающих действий, сопутствующих операциям, выполняемым в блоке `try`. Например:

```
f = open('foo','r')
try:
    # Выполнить некоторые действия
```

```
        ...
    finally:
        f.close()
        # Файл будет закрыт, независимо от того, что произойдет
```

Блок `finally` не используется для обработки ошибок. Он используется для реализации действий, которые должны выполняться всегда, независимо от того, возникла ошибка или нет. Если в блоке `try` исключений не возникло, блок `finally` будет выполнен сразу же вслед за ним. Если возникло исключение, управление сначала будет передано первой инструкции в блоке `finally`, а затем это исключение будет возбуждено повторно, чтобы обеспечить возможность его обработки в другом обработчике.

Встроенные типы исключений

В табл. 5.1 перечислены встроенные типы исключений, которые определены в языке Python.

Таблица 5.1. Встроенные типы исключений

Исключение	Описание
`BaseException`	Базовый класс всех исключений
`GeneratorExit`	Возбуждается методом `.close()` генераторов
`KeyboardInterrupt`	Возбуждается нажатием клавишей прерывания (обычно Ctrl-C)
`SystemExit`	Завершение программы
`Exception`	Базовый класс для всех исключений, не связанных с завершением программы
`StopIteration`	Возбуждается для прекращения итераций
`StandardError`	Базовый класс для всех встроенных исключений (только в Python 2). В Python 3 – базовый класс всех исключений, наследующих класс `Exception`
`ArithmeticError`	Базовый класс исключений, возбуждаемых арифметическими операциями
`FloatingPointError`	Ошибка операции с плавающей точкой
`ZeroDivisionError`	Деление или деления по модулю на ноль
`AssertionError`	Возбуждается инструкциями `assert`
`AttributeError`	Возбуждается при обращении к несуществующему атрибуту
`EnvironmentError`	Ошибка, обусловленная внешними причинами
`IOError`	Ошибка ввода-вывода при работе с файлами
`OSError`	Ошибка операционной системы
`EOFError`	Возбуждается по достижении конца файла

Таблица 5.1 (продолжение)

Исключение	Описание
ImportError	Ошибка в инструкции import
LookupError	Ошибка обращения по индексу или ключу
IndexError	Ошибка обращения по индексу за пределами последовательности
KeyError	Ошибка обращения к несуществующему ключу словаря
MemoryError	Нехватка памяти
NameError	Не удалось отыскать локальное или глобальное имя
UnboundLocalError	Ошибка обращения к локальной переменной, которой еще не было присвоено значение
ReferenceError	Ошибка обращения к объекту, который уже был уничтожен
RuntimeError	Универсальное исключение
NotImplementedError	Обращение к нереализованному методу или функции
SyntaxError	Синтаксическая ошибка
IndentationError	Ошибка оформления отступов
TabError	Непоследовательное использование символа табуляции (генерируется при запуске интерпретатора с ключом -tt)
SystemError	Нефатальная системная ошибка в интерпретаторе
TypeError	Попытка выполнить операцию над аргументом недопустимого типа
ValueError	Недопустимый тип
UnicodeError	Ошибка при работе с символами Юникода
UnicodeDecodeError	Ошибка декодирования символов Юникода
UnicodeEncodeError	Ошибка кодирования символов Юникода
UnicodeTranslateError	Ошибка трансляции символов Юникода

Исключения организованы в иерархию, как показано в табл. 5.1. Все исключения, входящие в ту или иную группу, могут быть перехвачены при использовании имени группы в определении блока except. Например:

```
try:
    инструкции
except LookupError: # Перехватит исключение IndexError или KeyError
    инструкции
```

или

```
try:
    инструкции
except Exception: # Перехватит любые программные исключения
    инструкции
```

Наверху иерархии исключения сгруппированы в зависимости от того, связаны ли они с завершением программы. Например, исключения `SystemExit` и `KeyboardInterrupt` не входят в группу `Exception`, потому что, когда программист предусматривает обработку всех программных исключений, он обычно не стремится перехватить завершение программы.

Определение новых исключений

Все встроенные исключения определяются как классы. Чтобы создать исключение нового типа, нужно объявить новый класс, наследующий класс `Exception`, как показано ниже:

```
class NetworkError(Exception): pass
```

Чтобы воспользоваться новым исключением, его достаточно просто передать инструкции `raise`, например:

```
raise NetworkError("Невозможно найти компьютер в сети.")
```

При возбуждении исключения с помощью инструкции `raise` допускается передавать конструктору класса дополнительные значения. В большинстве случаев это обычная строка, содержащая текст сообщения об ошибке. Однако пользовательские исключения могут предусматривать возможность приема одного или более значений, как показано в следующем примере:

```
class DeviceError(Exception):
    def __init__(self,errno,msg):
        self.args = (errno, msg)
        self.errno = errno
        self.errmsg = msg

# Возбудить исключение (передав несколько аргументов)
raise DeviceError(1, 'Нет ответа')
```

При создании собственного класса исключения, переопределяющего метод `__init__()`, важно не забыть присвоить кортеж, содержащий аргументы метода `__init__()`, атрибуту `self.args`, как было показано выше. Этот атрибут используется при выводе трассировочной информации. Если оставить этот атрибут пустым, пользователи не смогут увидеть информацию об исключении, когда возникнет ошибка.

Исключения могут быть организованы в иерархии с помощью механизма наследования. Например, исключение `NetworkError`, объявленное выше, могло бы служить базовым классом для более специфичных исключений. Например:

```
class HostnameError(NetworkError): pass
class TimeoutError(NetworkError): pass
```

```
def error1():
    raise HostnameError("Хост не найден")

def error2():
    raise TimeoutError("Превышено время ожидания")
try:
    error1()
except NetworkError as e:
    if type(e) is HostnameError:
        # Выполнить действия, характерные для ошибки этого типа
        ...
```

В данном случае инструкция `except NetworkError` перехватывает все исключения, наследующие класс `NetworkError`. Чтобы определить конкретный тип возникшей ошибки, выполняется проверка типа исключения с помощью функции `type()`. Для получения информации о последнем возникшем исключении можно также использовать функцию `sys.exc_info()`.

Менеджеры контекста и инструкция with

Надлежащее управление системными ресурсами, такими как файлы, блокировки и соединения, часто является достаточно сложной задачей, особенно в соединении с обработкой исключений. Например, возникшее исключение может заставить программу пропустить инструкции, отвечающие за освобождение критических ресурсов, таких как блокировки.

Инструкция `with` позволяет организовать выполнение последовательности инструкций внутри контекста, управляемого объектом, который играет роль менеджера контекста. Например:

```
with open("debuglog","a") as f:
    f.write("Отладка\n")
    инструкции
    f.write("Конец\n")

import threading
lock = threading.Lock()
with lock:
    # Начало критического блока
    инструкции
    # Конец критического блока
```

В первом примере инструкция `with` автоматически закроет открытый файл, когда поток управления покинет блок инструкций, следующий ниже. Во втором примере инструкция `with` автоматически захватит блокировку, когда поток управления войдет в блок инструкций, следующий ниже, и освободит ее при выходе.

Инструкция `with` *obj* позволяет объекту *obj* управлять происходящим, когда поток управления входит и покидает блок инструкций, следующий ниже. Когда интерпретатор встречает инструкцию `with` *obj*, он вызывает метод *obj*`.__enter__()`, чтобы известить объект о том, что был выполнен вход в новый контекст. Когда поток управления покидает контекст, вызывается метод *obj*`.__exit__(`*type*`,`*value*`,`*traceback*`)`. Если при выполнении инструкций

в блоке не возникло никаких исключений, во всех трех аргументах методу `__exit__()` передается значение `None`. В противном случае в них записываются тип исключения, его значение и трассировочная информация об исключении, вынудившем поток управления покинуть контекст. Метод `__exit__()` возвращает `True` или `False`, чтобы показать, было обработано исключение или нет (если возвращается значение `False`, возникшее исключение продолжит свое движение вверх за пределами контекста).

Инструкция `with` *obj* может принимать дополнительный спецификатор `as` *var*. В этом случае значение, возвращаемое методом *obj*.`__enter__()`, записывается в переменную *var*. Важно отметить, что в переменную *var* не обязательно будет записано значение *obj*.

Инструкция `with` способна работать только с объектами, поддерживающими протокол управления контекстом (методы `__enter__()` и `__exit__()`). Пользовательские классы также могут определять эти методы, чтобы реализовать собственный способ управления контекстом. Ниже приводится простой пример такого класса:

```
class ListTransaction(object):
    def __init__(self,thelist):
        self.thelist = thelist
    def __enter__(self):
        self.workingcopy = list(self.thelist)
        return self.workingcopy
    def __exit__(self,type,value,tb):
        if type is None:
            self.thelist[:] = self.workingcopy
        return False
```

Этот класс позволяет выполнять серию модификаций в существующем списке. При этом модификации вступят в силу, только если в процессе их выполнения не возникло исключения. В противном случае список останется в первоначальном состоянии. Например:

```
items = [1,2,3]
with ListTransaction(items) as working:
    working.append(4)
    working.append(5)
print(items)       # Выведет [1,2,3,4,5]

try:
    with ListTransaction(items) as working:
        working.append(6)
        working.append(7)
        raise RuntimeError("Немножко смошенничаем!")
except RuntimeError:
    pass
print(items) # Выведет [1,2,3,4,5]
```

Модуль `contextlib` упрощает реализацию собственных менеджеров контекста за счет обертывания функций-генераторов. Например:

```
from contextlib import contextmanager
@contextmanager
```

```
def ListTransaction(thelist):
    workingcopy = list(thelist)
    yield workingcopy
    # Изменить оригинальный список, только если не возникло ошибок
    thelist[:] = workingcopy
```

В этом примере значение, передаваемое инструкции `yield`, используется, как возвращаемое значение метода `__enter__()`. После вызова метода `__exit__()` выполнение будет продолжено с инструкции, следующей за инструкцией `yield`. Если в контексте возникло исключение, оно проявится как исключение в функции-генераторе. При необходимости функция может перехватить и обработать исключение, но в данном случае оно продолжит свое распространение за пределы генератора и, возможно, будет обработано где-то в другом месте.

Отладочные проверки и переменная __debug__

Инструкция `assert` позволяет добавлять в программу отладочный код. В общем случае инструкция `assert` имеет следующий вид:

```
assert test [, msg]
```

где *test* – выражение, которое должно возвращать значение `True` или `False`. Если выражение *test* возвратит значение `False`, инструкция `assert` возбудит исключение `AssertionError` с переданным ему сообщением *msg*. Например:

```
def write_data(file,data):
    assert file, "write_data: файл не определен!"
...
```

Инструкция `assert` не должна содержать программный код, обеспечивающий безошибочную работу программы, потому что он не будет выполняться интерпретатором, работающим в оптимизированном режиме (этот режим включается при запуске интерпретатора с ключом `-O`). В частности, будет ошибкой использовать инструкцию `assert` для проверки ввода пользователя. Обычно инструкции `assert` используются для проверки условий, которые всегда должны быть истинными; если такое условие нарушается, это можно рассматривать, как ошибку в программе, а не как ошибку пользователя.

Например, если функция `write_data()` в примере выше была бы предназначена для взаимодействия с конечным пользователем, инструкцию `assert` следовало бы заменить обычной условной инструкцией `if` и предусмотреть обработку ошибок.

Помимо инструкции `assert` в Python имеется встроенная переменная `__debug__`, доступная только для чтения, которая получает значение `True`, когда интерпретатор работает не в оптимизированном режиме (включается при запуске интерпретатора с ключом `-O`). Программы могут проверять значение этой переменной, чтобы в случае необходимости выполнять дополнительную проверку на наличие ошибок. Внутренняя реализация перемен-

ной __debug__ в интерпретаторе оптимизирована так, что сама инструкция if в действительности не включается в текст программы. Если интерпретатор действует в обычном режиме, программный код, составляющий тело инструкции if __debug__, просто включается в текст программы, без самой инструкции if. При работе в оптимизированном режиме инструкция if __debug__ и все связанные с ней инструкции полностью удаляются из программы.

Использование инструкции assert и переменной __debug__ обеспечивает возможность эффективной разработки программ, работающих в двух режимах. Например, для работы в отладочном режиме можно добавить в свой программный код произвольное количество проверок, чтобы убедиться в его безошибочной работе. При работе в оптимизированном режиме все эти проверки будут удалены и уже не будут отрицательно влиять на производительность.

6

Функции и функциональное программирование

В большинстве своем программы разбиты на функции – для достижения большей модульности и упрощения обслуживания. Язык Python не только облегчает объявление функций, но и снабжает их удивительным количеством дополнительных особенностей, унаследованных из функциональных языков программирования. В этой главе описываются функции, правила видимости, замыкания, декораторы, генераторы, сопрограммы и другие особенности, присущие функциональному программированию. Кроме того, здесь же описываются генераторы списков и выражения-генераторы, которые являются мощными инструментами программирования в декларативном стиле и обработки данных.

Функции

Функции объявляются с помощью инструкции `def`:

```
def add(x,y):
    return x + y
```

Тело функции – это простая последовательность инструкций, которые выполняются при вызове функции. Вызов функции осуществляется как обращение к имени функции, за которым следует кортеж аргументов, например: *a* `= add(3,4)`. Порядок следования аргументов в вызове должен совпадать с порядком следования аргументов в определении функции. Если будет обнаружено несоответствие, интерпретатор возбудит исключение `TypeError`.

Аргументы функции могут иметь значения по умолчанию. Например:

```
def split(line,delimiter=','):
    инструкции
```

Если в объявлении функции присутствует аргумент со значением по умолчанию, этот аргумент и все следующие за ним считаются необязательны-

ми. Если значения по умолчанию назначены не всем необязательным аргументам, возбуждается исключение SyntaxError.

Значения аргументов по умолчанию всегда связываются с теми самыми объектами, которые использовались в качестве значений в объявлении функции. Например:

```
a = 10
def foo(x=a):
    return x

a = 5          # Изменить значение 'a'.
foo()          # вернет 10 (значение по умолчанию не изменилось)
```

При этом использование изменяемых объектов в качестве значений по умолчанию может приводить к неожиданным результатам:

```
def foo(x, items=[]):
    items.append(x)
    return items

foo(1) # вернет [1]
foo(2) # вернет [1, 2]
foo(3) # вернет [1, 2, 3]
```

Обратите внимание, что аргумент по умолчанию запоминает изменения в результате предыдущих вызовов. Чтобы предотвратить такое поведение, в качестве значения по умолчанию лучше использовать None и добавить проверку, как показано ниже:

```
def foo(x, items=None):
    if items is None:
        items = []
    items.append(x)
    return items
```

Если перед последним аргументом в определении функции добавить символ звездочки (*), функция сможет принимать переменное число аргументов:

```
def fprintf(file, fmt, *args):
    file.write(fmt % args)

# Вызов функции fprintf.
# Аргумент args получит значение (42,"hello world", 3.45)
fprintf(out,"%d %s %f", 42, "hello world", 3.45)
```

В данном случае все дополнительные аргументы будут помещены в переменную *args* в виде кортежа. Чтобы передать кортеж *args* другой функции, как если бы это был набор аргументов, достаточно использовать синтаксис *args, как показано ниже:

```
def printf(fmt, *args):
    # Вызвать другую функцию и передать ей аргумент args
    fprintf(sys.stdout, fmt, *args)
```

Кроме того, имеется возможность передавать функциям аргументы, явно указывая их имена и значения. Такие аргументы называются *именованными аргументами*. Например:

```
def foo(w,x,y,z):
    инструкции

# Вызов с именованными аргументами
foo(x=3, y=22, w='hello', z=[1,2])
```

При вызове с именованными аргументами порядок следования аргументов не имеет значения. Однако при этом должны быть явно указаны имена всех обязательных аргументов, если только они не имеют значений по умолчанию. Если пропустить какой-либо из обязательных аргументов или использовать именованный аргумент, не совпадающий ни с одним из имен параметров в определении функции, будет возбуждено исключение `TypeError`. Кроме того, так как в языке **Python любая функция может быть вызвана** с использованием именованных аргументов, то резонно придерживаться правила – объявлять функции с описательными именами аргументов.

В одном вызове функции допускается одновременно использовать позиционные и именованные аргументы, при соблюдении следующих условий: первыми должны быть указаны позиционные аргументы, должны быть определены значения для всех обязательных аргументов и ни для одного из аргументов не должно быть передано более одного значения. Например:

```
foo('hello', 3, z=[1,2], y=22)
foo(3, 22, w='hello', z=[1,2]) # TypeError. Несколько значений для w
```

Если последний аргумент в объявлении функции начинается с символов `**`, все дополнительные именованные аргументы (имена которых отсутствуют в объявлении функции) будут помещены в словарь и переданы функции. Это обеспечивает удобную возможность писать функции, способные принимать значительное количество параметров, описание которых в объявлении функции выглядело бы слишком громоздко. Например:

```
def make_table(data, **parms):
    # Получить параметры из аргумента parms (словарь)
    fgcolor = parms.pop("fgcolor","black")
    bgcolor = parms.pop("bgcolor","white")
    width = parms.pop("width",None)
    ...
    # Нет больше параметров
    if parms:
        raise TypeError("Неподдерживаемые параметры %s" % list(parms))

make_table(items, fgcolor="black", bgcolor="white", border=1,
           borderstyle="grooved", cellpadding=10,
           width=400)
```

Допускается одновременно использовать дополнительные именованные аргументы и список позиционных аргументов переменной длины, при условии, что аргумент, начинающийся с символов `**`, указан последним:

```
# Принимает переменное количество позиционных или именованных аргументов
def spam(*args, **kwargs):
    # args – кортеж со значениями позиционных аргументов
    # kwargs – словарь с именованными аргументами
    ...
```

Словарь с дополнительными именованными аргументами также можно передать другой функции, используя синтаксис `**kwargs`:

```
def callfunc(*args, **kwargs):
    func(*args,**kwargs)
```

Такой способ использования аргументов `*args` и `**kwargs` часто применяется при создании оберток для других функций. Например, функция `callfunc()` принимает любые комбинации аргументов и просто передает их функции `func()`.

Передача параметров и возвращаемые значения

Параметры функции, которые передаются ей при вызове, являются обычными именами, ссылающимися на входные объекты. Семантика передачи параметров в языке Python не имеет точного соответствия какому-либо одному способу, такому как «передача по значению» или «передача по ссылке», которые могут быть вам знакомы по другим языкам программирования. Например, если функции передается неизменяемое значение, это выглядит, как передача аргумента по значению. Однако при передаче изменяемого объекта (такого как список или словарь), который модифицируется функцией, эти изменения будут отражаться на исходном объекте. Например:

```
a = [1, 2, 3, 4, 5]
def square(items):
    for i,x in enumerate(items):
        items[i] = x * x       # Изменяется элемент в самом списке

square(a)                      # Изменит содержимое списка: [1, 4, 9, 16, 25]
```

Если функция изменяет значения аргументов или оказывает влияние на состояние других частей программы, про такие функции говорят, что они имеют *побочные эффекты*. В общем случае подобного стиля программирования лучше избегать, потому что с ростом размеров или сложности программы такие функции могут стать источником трудноуловимых ошибок (из инструкции вызова функции сложно определить, имеет ли она побочные эффекты). Такие функции трудно приспособить для работы в составе многопоточных программ, потому что побочные эффекты обычно приходится защищать блокировками.

Инструкция `return` возвращает значение из функции. Если значение не указано или опущена сама инструкция `return`, вызывающей программе возвращается объект `None`. Чтобы вернуть несколько значений, достаточно поместить их в кортеж:

```
def factor(a):
    d = 2
    while (d <= (a / 2)):
        if ((a / d) * d == a):
            return ((a / d), d)
        d = d + 1
    return (a, 1)
```

Если функция возвращает несколько значений в виде кортежа, их можно присвоить сразу нескольким отдельным переменным:

```
x, y = factor(1243)   # Возвращаемые значения записываются в переменные x и y.
```

или

```
(x, y) = factor(1243) # Альтернативная версия. Результат тот же самый.
```

Правила видимости

При каждом вызове функции создается новое локальное пространство имен. Это пространство имен представляет локальное окружение, содержащее имена параметров функции, а также имена переменных, которым были присвоены значения в теле функции. Когда возникает необходимость отыскать имя, интерпретатор в первую очередь просматривает локальное пространство имен. Если искомое имя не было найдено, поиск продолжается в глобальном пространстве имен. Глобальным пространством имен для функций всегда является пространство имен модуля, в котором эта функция была определена. Если интерпретатор не найдет искомое имя в глобальном пространстве имен, поиск будет продолжен во встроенном пространстве имен. Если и эта попытка окажется неудачной, будет возбуждено исключение `NameError`.

Одна из необычных особенностей пространств имен заключается в работе с глобальными переменными внутри функции. Рассмотрим в качестве примера следующий фрагмент:

```
a = 42
def foo():
    a = 13
foo()
# Переменная a по-прежнему имеет значение 42
```

После выполнения этого фрагмента переменная `a` по-прежнему будет иметь значение `42`, несмотря на то что внутри функции `foo` значение переменной `a` изменяется. Когда внутри функции выполняется операция присваивания значения переменной, она всегда выполняется в локальном пространстве имен функции; в результате переменная `a` в теле функции ссылается на совершенно другой объект, содержащий значение `13`, а не на тот, на который ссылается внешняя переменная. Обеспечить иное поведение можно с помощью инструкции `global`. Она просто объявляет принадлежность имен глобальному пространству имен; использовать ее необходимо, только когда потребуется изменить глобальную переменную. Ее можно поместить где-нибудь в теле функции и использовать неоднократно. Например:

```
a = 42
b = 37
def foo():
    global a # переменная 'a' находится в глобальном пространстве имен
    a = 13
    b = 0
foo()
# теперь a имеет значение 13. b - по-прежнему имеет значение 37.
```

В языке **Python поддерживается возможность определять вложенные функции. Например:**

```
def countdown(start):
    n = start
    def display(): # Объявление вложенной функции
        print('T-minus %d' % n)
    while n > 0:
        display()
        n -= 1
```

Переменные во вложенных функциях привязаны к *лексической области видимости*. То есть поиск имени переменной начинается в локальной области видимости и затем последовательно продолжается во всех объемлющих областях видимости внешних функций, в направлении от внутренних к внешним. Если и в этих пространствах имен искомое имя не будет найдено, поиск будет продолжен в глобальном, а затем во встроенном пространстве имен, как и прежде. Несмотря на доступность промежуточных вмещающих областей видимости, **Python 2 позволяет выполнять присваивание только переменным, находящимся в самой внутренней (локальные переменные) и в самой внешней (при использовании инструкции** `global`**) области видимости. По этой причине вложенные функции не имеют возможности присваивать значения локальным переменным, определенным во внешних функциях. Например, следующий программный код не будет работать:**

```
def countdown(start):
    n = start
    def display():
        print('T-minus %d' % n)
    def decrement():
        n -= 1 # Не будет работать в Python 2
    while n > 0:
        display()
        decrement()
```

В Python 2 эту проблему можно решить, поместив значения, которые предполагается изменять, в список или в словарь. В Python 3 можно объявить переменную `n` **как** `nonlocal`**, например:**

```
def countdown(start):
    n = start
    def display():
        print('T-minus %d' % n)
    def decrement():
```

```
        nonlocal n  # Связывает имя с внешней переменной n (только в Python 3)
        n -= 1
    while n > 0:
        display()
        decrement()
```

Объявление `nonlocal` не может использоваться для привязки указанного имени к локальной переменной, определенной внутри одной из вложенных функций (то есть в *динамической области видимости*). Поэтому для тех, кто имеет опыт работы с языком Perl, замечу, что объявление `nonlocal` – это не то же самое, что объявление `local` в языке Perl.

При обращении к локальной переменной до того, как ей будет присвоено значение, возбуждается исключение `UnboundLocalError`. Следующий пример демонстрирует один из возможных сценариев, когда такое исключение может возникнуть:

```
i = 0
def foo():
    i = i + 1     # Приведет к исключению UnboundLocalError
    print(i)
```

В этой функции переменная `i` определяется как локальная (потому что внутри функции ей присваивается некоторое значение и отсутствует инструкция `global`). При этом инструкция присваивания `i = i + 1` пытается прочитать значение переменной `i` еще до того, как ей будет присвоено значение. Хотя в этом примере существует глобальная переменная `i`, она не используется для получения значения. Переменные в функциях могут быть либо локальными, либо глобальными и не могут произвольно изменять область видимости в середине функции. Например, нельзя считать, что переменная `i` в выражении `i + 1` в предыдущем фрагменте обращается к глобальной переменной `i`; при этом переменная `i` в вызове `print(i)` подразумевает локальную переменную `i`, созданную в предыдущей инструкции.

Функции как объекты и замыкания

Функции в языке Python – объекты первого класса. Это означает, что они могут передаваться другим функциям в виде аргументов, сохраняться в структурах данных и возвращаться функциями в виде результата. Ниже приводится пример функции, которая на входе принимает другую функцию и вызывает ее:

```
# foo.py
def callf(func):
    return func()
```

А это пример использования функции, объявленной выше:

```
>>> import foo
>>> def helloworld():
...     return 'Привет, Мир!'
...
>>> foo.callf(helloworld)    # Передача функции в виде аргумента
```

```
'Привет, Мир!'
>>>
```

Когда функция интерпретируется как данные, она неявно несет информацию об окружении, в котором была объявлена функция, что оказывает влияние на связывание свободных переменных в функции. В качестве примера рассмотрим модифицированную версию файла `foo.py`, в который были добавлены переменные:

```
# foo.py
x = 42
def callf(func):
    return func()
```

Теперь исследуем следующий пример:

```
>>> import foo
>>> x = 37
>>> def helloworld():
...     return "Привет, Мир! x = %d" % x
...
>>> foo.callf(helloworld)     # Передача функции в виде аргумента
'Привет, Мир! x = 37'
>>>
```

Обратите внимание, как функция `helloworld()` в этом примере использует значение переменной `x`, которая была определена в том же окружении, что и сама функция `helloworld()`. Однако хотя переменная `x` определена в файле `foo.py` и именно там фактически вызывается функция `helloworld()`, при исполнении функцией `helloworld()` используется не это значение переменной `x`.

Когда инструкции, составляющие функцию, упаковываются вместе с окружением, в котором они выполняются, получившийся объект называют *замыканием*. Такое поведение предыдущего примера объясняется наличием у каждой функции атрибута `__globals__`, ссылающегося на глобальное пространство имен, в котором функция была определена. Это пространство имен всегда соответствует модулю, в котором была объявлена функция. Для предыдущего примера атрибут `__globals__` содержит следующее:

```
>>> helloworld.__globals__
{'__builtins__': <module '__builtin__' (built-in)>,
 'helloworld': <function helloworld at 0x7bb30>,
 'x': 37, '__name__': '__main__', '__doc__': None
 'foo': <module 'foo' from 'foo.py'>}
>>>
```

Когда функция используется как вложенная, в замыкание включается все ее окружение, необходимое для работы внутренней функции. Например:

```
import foo
def bar():
    x = 13
    def helloworld():
        return "Привет, Мир! x = %d" % x
    foo.callf(helloworld) # вернет 'Привет, Мир! x = 13'
```

Замыкания и вложенные функции особенно удобны, когда требуется написать программный код, реализующий концепцию отложенных вычислений. Рассмотрим еще один пример:

```
from urllib import urlopen
# from urllib.request import urlopen (Python 3)
def page(url):
    def get():
        return urlopen(url).read()
    return get
```

Функция `page()` в этом примере не выполняет никаких вычислений. Она просто создает и возвращает функцию `get()`, которая при вызове будет извлекать содержимое веб-страницы. То есть вычисления, которые производятся в функции `get()`, в действительности откладываются до момента, когда фактически будет вызвана функция `get()`. Например:

```
>>> python = page("http://www.python.org")
>>> jython = page("http://www.jython.org")
>>> python
<function get at 0x95d5f0>
>>> jython
<function get at 0x9735f0>
>>> pydata = python() # Извлечет страницу http://www.python.org
>>> jydata = jython() # Извлечет страницу http://www.jython.org
>>>
```

Две переменные, `python` и `jython`, объявленные в этом примере, в действительности являются двумя различными версиями функции `get()`. Хотя функция `page()`, которая создала эти значения, больше не выполняется, тем не менее обе версии функции `get()` неявно несут в себе значения внешних переменных на момент создания функции `get()`. То есть при выполнении функция `get()` вызовет `urlopen(url)` со значением `url`, которое было передано функции `page()`. Взглянув на атрибуты объектов `python` и `jython`, можно увидеть, какие значения переменных были включены в замыкания. Например:

```
>>> python.__closure__
(<cell at 0x67f50: str object at 0x69230>,)
>>> python.__closure__[0].cell_contents
'http://www.python.org'
>>> jython.__closure__[0].cell_contents
'http://www.jython.org'
>>>
```

Замыкание может быть весьма эффективным способом сохранения информации о состоянии между вызовами функции. Например, рассмотрим следующий пример, в котором реализован простой счетчик:

```
def countdown(n):
    def next():
        nonlocal n
        r = n
        n -= 1
        return r
    return next
```

```
# Пример использования
next = countdown(10)
while True:
    v = next() # Получить следующее значение
    if not v: break
```

В этом примере для хранения значения внутреннего счетчика `n` используется замыкание. Вложенная функция `next()` при каждом вызове уменьшает значение счетчика и возвращает его предыдущее значение. Программисты, незнакомые с замыканиями, скорее всего реализовали бы такой счетчик с помощью класса, например:

```
class Countdown(object):
    def __init__(self,n):
        self.n = n
    def next(self):
        r = self.n
        self.n -= 1
        return r

# Пример использования
c = Countdown(10)
while True:
    v = c.next() # Получить следующее значение
    if not v: break
```

Однако если увеличить начальное значение обратного счетчика и произвести простейшие измерения производительности, можно обнаружить, что версия, основанная на замыканиях, выполняется значительно быстрее (на машине автора прирост скорости составил почти 50%).

Тот факт, что замыкания сохраняют в себе окружение вложенных функций, делает их удобным инструментом, когда требуется обернуть существующую функцию с целью расширить ее возможности. Об этом рассказывается в следующем разделе.

Декораторы

Декоратор – это функция, основное назначение которой состоит в том, чтобы служить оберткой для другой функции или класса. Главная цель такого обертывания – изменить или расширить возможности обертываемого объекта. Синтаксически декораторы оформляются добавлением специального символа `@` к имени, как показано ниже:

```
@trace
def square(x):
    return x*x
```

Предыдущий фрагмент является сокращенной версией следующего фрагмента:

```
def square(x):
    return x*x
square = trace(square)
```

В этом примере объявляется функция square(). Однако сразу же вслед за объявлением объект функции передается функции trace(), а возвращаемый ею объект замещает оригинальный объект square. Теперь рассмотрим реализацию функции trace, чтобы выяснить, что полезного она делает:

```
enable_tracing = True
if enable_tracing:
    debug_log = open("debug.log","w")

def trace(func):
    if enable_tracing:
        def callf(*args,**kwargs):
            debug_log.write("Вызов %s: %s, %s\n" %
                            (func.__name__, args, kwargs))
            r = func(*args,**kwargs)
            debug_log.write("%s вернула %s\n" % (func.__name, r))
            return r
        return callf
    else:
        return func
```

В этом фрагменте функция trace() создает функцию-обертку, которая записывает некоторую отладочную информацию в файл и затем вызывает оригинальный объект функции. То есть, если теперь вызвать функцию square(), в файле debug.log можно будет увидеть результат вызова метода write() в функции-обертке. Функция callf, которая возвращается функцией trace(), — это замыкание, которым замещается оригинальная функция. Но самое интересное в этом фрагменте заключается в том, что возможность трассировки включается только с помощью глобальной переменной enable_tracing. Если этой переменной присвоить значение False, декоратор trace() просто вернет оригинальную функцию, без каких-либо изменений. То есть, когда трассировка отключена, использование декоратора не влечет за собой снижения производительности.

При использовании декораторы должны помещаться в отдельной строке, непосредственно перед объявлением функции или класса. Кроме того, допускается указывать более одного декоратора. Например:

```
@foo
@bar
@spam
def grok(x):
    pass
```

В этом случае декораторы применяются в порядке следования. Результат получается тот же самый, что и ниже:

```
def grok(x):
    pass
grok = foo(bar(spam(grok)))
```

Кроме того, декоратор может принимать аргументы. Например:

```
@eventhandler('BUTTON')
def handle_button(msg):
    ...
```

```
@eventhandler('RESET')
def handle_reset(msg):
    ...
```

При наличии аргументов декораторы имеют следующую семантику:

```
def handle_button(msg):
...
temp = eventhandler('BUTTON')       # Вызов декоратора с аргументами
handle_button = temp(handle_button) # Вызов функции, возвращаемой декоратором
```

В этом случае функция декоратора со спецификатором @ просто принимает аргументы. Затем она возвращает функцию, которая вызывается с декорируемой функцией в виде аргумента. Например:

```
# Декоратор обработчика события
event_handlers = { }
def eventhandler(event):
    def register_function(f):
        event_handlers[event] = f
        return f
    return register_function
```

Декораторы могут также применяться к определениям классов. Например:

```
@foo
class Bar(object):
    def __init__(self,x):
        self.x = x
    def spam(self):
        инструкции
```

Функция-декоратор, применяемая к классу, всегда должна возвращать объект класса. Программному коду, действующему с оригинальным классом, может потребоваться напрямую обращаться к методам и атрибутам класса, например: `Bar.spam`. Это будет невозможно, если функция-декоратор `foo()` будет возвращать функцию.

Декораторы могут оказывать нежелательное воздействие на такие аспекты функций, как рекурсия, строки документирования и атрибуты. Эти проблемы описываются ниже в этой главе.

Генераторы и инструкция yield

Если функция использует ключевое слово `yield`, тем самым объявляется объект, который называется *генератором*. Генератор – это функция, которая воспроизводит последовательность значений и может использоваться при выполнении итераций. Например:

```
def countdown(n):
    print("Обратный отсчет, начиная с %d" % n)
    while n > 0:
        yield n
        n -= 1
    return
```

Если вызвать эту функцию, можно заметить, что ни одна инструкция в ее теле не будет выполнена. Например:

```
>>> c = countdown(10)
>>>
```

Вместо выполнения функции интерпретатор создает и возвращает объект-генератор. Объект-генератор, в свою очередь, выполняет функцию, когда вызывается его метод `next()` (или `__next__()`, в Python 3). Например:

```
>>> c.next()     # В Python 3 нужно вызвать метод c.__next__()
Обратный отсчет, начиная с 10
10
>>> c.next()
9
```

При вызове метода `next()` инструкции функции-генератора выполняются, пока не будет встречена инструкция `yield`. Инструкция `yield` возвращает результат, и выполнение функции приостанавливается в этой точке, пока снова не будет вызван метод `next()`. После этого выполнение функции возобновляется, начиная с инструкции, следующей за инструкцией `yield`.

Как правило, в обычной ситуации не приходится непосредственно вызывать метод `next()` генератора, потому что чаще всего генераторы используются в инструкции `for`, в функции `sum()` и некоторых других операциях над последовательностями. Например:

```
for n in countdown(10):
    инструкции
a = sum(countdown(10))
```

Функция-генератор сигнализирует о завершении последовательности значений и прекращении итераций, возбуждая исключение `StopIteration`. Для генераторов считается недопустимым по завершении итераций возвращать значение, отличное от `None`.

С функциями-генераторами связана одна трудноуловимая проблема, когда последовательность, создаваемая такой функцией, извлекается только частично. Например, рассмотрим следующий фрагмент:

```
for n in countdown(10):
    if n == 2: break
    statements
```

В этом примере цикл `for` прерывается инструкцией `break`, и ассоциированный с ней генератор никогда не достигнет конца генерируемой последовательности. Для решения этой проблемы объекты-генераторы предоставляют метод `close()`, который используется, чтобы известить объект о завершении итераций. Когда генератор больше не используется или удаляется, вызывается метод `close()`. Обычно нет необходимости вызывать `close()`, тем не менее его можно вызвать вручную, как показано ниже:

```
>>> c = countdown(10)
>>> c.next()
Обратный отсчет, начиная с 10
10
```

```
>>> c.next()
9
>>> c.close()
>>> c.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>>
```

При вызове метода `close()` внутри функции-генератора, в инструкции `yield`, возбуждается исключение `GeneratorExit`. При необходимости это исключение можно перехватить и выполнить завершающие действия.

```
def countdown(n):
    print("Обратный отсчет, начиная с %d" % n)
    try:
        while n > 0:
            yield n
            n = n - 1
    except GeneratorExit:
        print("Достигнуто значение %d" % n)
```

Несмотря на имеющуюся возможность перехватить исключение `GeneratorExit`, для функций-генераторов считается недопустимым обрабатывать исключения и продолжать вывод других значений с помощью инструкции `yield`. Кроме того, когда программа выполняет итерации по значениям, возвращаемым генератором, не следует асинхронно вызывать метод `close()` этого генератора в другом потоке выполнения или в обработчике сигналов.

Сопрограммы и выражения yield

Внутри функций инструкция `yield` может также использоваться как выражение, стоящее справа от оператора присваивания. Например:

```
def receiver():
    print("Готов к приему значений")
    while True:
        n = (yield)
        print("Получено %s" % n)
```

Функция, использующая инструкцию `yield` таким способом, называется *сопрограммой* и выполняется в ответ на попытку передать ей значение. Своим поведением такие функции очень похожи на генераторы. Например:

```
>>> r = receiver()
>>> r.next() # Выполнить до первой инструкции yield (r.__next__() в Python 3)
Готов к приему значений
>>> r.send(1)
Получено 1
>>> r.send(2)
Получено 2
>>> r.send("Привет")
Получено Привет
>>>
```

В этом примере первый вызов `next()` необходим, чтобы выполнить инструкции в теле сопрограммы, предшествующие первому выражению `yield`. В этой точке выполнение сопрограммы приостанавливается, пока ей не будет отправлено значение с помощью метода `send()` объекта-генератора `r`. Значение, переданное методу `send()`, возвращается выражением `(yield)` в теле сопрограммы. После получения значения сопрограмма продолжает выполнение, пока не будет встречена следующая инструкция `yield`.

Программисты часто допускают ошибку, забывая о необходимости инициирующего вызова метода `next()` сопрограммы. По этой причине рекомендуется оборачивать сопрограммы декоратором, который автоматически выполняет этот шаг.

```
def coroutine(func):
    def start(*args,**kwargs):
        g = func(*args,**kwargs)
        g.next()
        return g
    return start
```

При наличии этого декоратора его можно было бы применить к сопрограммам:

```
@coroutine
def receiver():
    print("Готов к приему значений")
    while True:
        n = (yield)
        print("Получено %s" % n)

# Пример использования
r = receiver()
r.send("Привет, Мир!") # Внимание: начальный вызов .next() не требуется
```

Сопрограмма обычно может выполняться до бесконечности, пока она явно не будет остановлена или пока не завершится сама. Закрыть поток входных данных можно вызовом метода `close()`, например:

```
>>> r.close()
>>> r.send(4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

Если после завершения сопрограммы попытаться передать ей значения, интерпретатор возбудит исключение `StopIteration`. Вызов метода `close()` возбуждает исключение `GeneratorExit` внутри сопрограммы, как уже было описано в разделе о генераторах. Например:

```
def receiver():
    print("Готов к приему значений")
    try:
        while True:
            n = (yield)
            print("Получено %s" % n)
```

```
        except GeneratorExit:
            print("Прием завершен")
```

Исключения внутри сопрограммы можно также возбуждать с помощью метода `throw(exctype [, value [, tb]])`, где *exctype* – тип исключения, *value* – значение исключения и *tb* объект с трассировочной информацией. Например:

```
>>> r.throw(RuntimeError,"Вас надули!")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 4, in receiver
RuntimeError: Вас надули!
```

Исключения возбуждаются так, как если бы их источником была инструкция `yield` в сопрограмме. Сопрограмма может перехватывать и обрабатывать такие исключения. Никогда не следует асинхронно вызывать метод `throw()`, из другого потока выполнения или из обработчика сигнала, так как это небезопасно.

Сопрограмма может одновременно принимать и возвращать значения, используя одну и ту же инструкцию `yield`, для чего достаточно добавить в выражение `yield` возвращаемое значение. Следующий пример иллюстрирует этот прием:

```
def line_splitter(delimiter=None):
    print("Все готово к разбиению строки")
    result = None
    while True:
        line = (yield result)
        result = line.split(delimiter)
```

В данном случае сопрограмма используется точно так же, как и прежде. Однако теперь метод `send()` возвращает результат. Например:

```
>>> s = line_splitter(",")
>>> s.next()
Все готово к разбиению строки
>>> s.send("A,B,C")
['A', 'B', 'C' ]
>>> s.send("100,200,300")
['100', '200', '300']
>>>
```

Понимание порядка операций, выполняемых в этом примере, имеет важное значение. Первый вызов метода `next()` выполняет сопрограмму до первого выражения (`yield result`), которое возвращает объект `None` – начальное значение переменной `result`. При последующих вызовах `send()` принимаемое значение помещается в переменную `line` и преобразуется в список, который сохраняется в переменной `result`. Значение, возвращаемое методом `send()`, – это значение, переданное следующей инструкции `yield`.

Другими словами, значение, возвращаемое методом `send()`, поступает от выражения `yield`, а не от инструкции, ответственной за прием значения, посылаемого сопрограмме с помощью метода `send()`. Если сопрограмма воз-

вращает какие-либо значения, необходимо предусмотреть в ней обработку исключений, возбуждаемых с помощью метода `throw()`. Если возбудить исключение вызовом метода `throw()`, то значение, переданное выражению `yield` в сопрограмме, будет возвращено методом `throw()`. Если вы забудете сохранить это значение, оно будет безвозвратно утрачено.

Использование генераторов и сопрограмм

На первый взгляд не совсем очевидно, как можно было бы использовать генераторы и сопрограммы для решения практических задач. Однако генераторы и сопрограммы могут быть весьма эффективным инструментом решения проблем в системном программировании, в разработке сетевых приложений или в реализации распределенных вычислений. Например, функции-генераторы могут использоваться для организации конвейерной обработки данных, подобной той, что широко используется в командной оболочке UNIX. Один из примеров такого использования генераторов приводится в главе 1 «Вводное руководство». Ниже приводится еще один пример, в котором задействовано несколько функций-генераторов, выполняющих поиск, открытие, чтение и обработку файлов:

```
import os
import fnmatch

def find_files(topdir, pattern):
    for path, dirname, filelist in os.walk(topdir):
        for name in filelist:
            if fnmatch.fnmatch(name, pattern):
                yield os.path.join(path,name)

import gzip, bz2
def opener(filenames):
    for name in filenames:
        if name.endswith(".gz"): f = gzip.open(name)
        elif name.endswith(".bz2"): f = bz2.BZ2File(name)
        else: f = open(name)
        yield f

def cat(filelist):
    for f in filelist:
        for line in f:
            yield line

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

Ниже приводится пример использования этих функций для организации конвейерной обработки:

```
wwwlogs = find("www","access-log*")
files   = opener(wwwlogs)
lines   = cat(files)
pylines = grep("python", lines)
```

```
for line in pylines:
    sys.stdout.write(line)
```

Программа, представленная в этом примере, обрабатывает все строки во всех файлах с именами "access-log*", присутствующих во всех подкаталогах, находящихся в корневом каталоге "www". Для каждого файла "access-log" выясняется, является ли он сжатым, после чего каждый из них открывается с помощью соответствующего механизма. Все строки объединяются вместе и обрабатываются фильтром, который отыскивает текст "python". Вся программа управляется инструкцией for в конце. Каждая итерация этого цикла извлекает новое значение из конвейера и выводит его. Кроме того, эта реализация отличается весьма низким потреблением памяти, потому что она не создает ни временных списков, ни каких-либо других структур данных большого размера.

Сопрограммы могут использоваться в программах, занимающихся обработкой потоков данных. Программы, организованные таким способом, напоминают перевернутые конвейеры. Вместо того, чтобы в цикле for извлекать значения из последовательности функций-генераторов, такие программы передают значения коллекции взаимосвязанных сопрограмм. Ниже приводится пример функций сопрограмм, имитирующих функции-генераторы, представленные выше:

```
import os
import fnmatch

@coroutine
def find_files(target):
    while True:
        topdir, pattern = (yield)
        for path, dirname, filelist in os.walk(topdir):
            for name in filelist:
                if fnmatch.fnmatch(name,pattern):
                    target.send(os.path.join(path,name))

import gzip, bz2
@coroutine
def opener(target):
    while True:
        name = (yield)
        if name.endswith(".gz"): f = gzip.open(name)
        elif name.endswith(".bz2"): f = bz2.BZ2File(name)
        else: f = open(name)
        target.send(f)

@coroutine
def cat(target):
    while True:
        f = (yield)
        for line in f:
            target.send(line)

@coroutine
def grep(pattern, target):
```

```
    while True:
        line = (yield)
        if pattern in line:
            target.send(line)

@coroutine
def printer():
    while True:
        line = (yield)
        sys.stdout.write(line)
```

Ниже показано, как можно связать все эти сопрограммы, чтобы создать конвейер для обработки потока данных:

```
finder = find_files(opener(cat(grep("python",printer()))))

# Теперь передать значение
finder.send(("www","access-log*"))
finder.send(("otherwww","access-log*"))
```

В этом примере каждая сопрограмма передает данные другой сопрограмме, указанной в аргументе `target`. В отличие от примера с генераторами, здесь все управление осуществляется передачей данных первой сопрограмме `find_files()`. Эта сопрограмма, в свою очередь, передает данные следующей стадии. Критическим аспектом этого примера является то, что конвейер может оставаться активным неопределенно долго или пока явно не будет вызван метод `close()`. Благодаря этому программа может продолжать передавать данные сопрограмме так долго, сколько потребуется, например, выполнить подряд два вызова `send()`, как показано в примере.

Сопрограммы могут использоваться для реализации своего рода многозадачности. Например, центральный диспетчер задач или цикл событий могут создавать и передавать данные огромным коллекциям из сотен или даже тысяч сопрограмм, выполняющих различные виды обработки. Тот факт, что входные данные «посылаются» сопрограмме, означает также, что сопрограммы легко могут внедряться в программы, использующие очереди сообщений и передачу данных для организации взаимодействий между различными компонентами программы. Дополнительные сведения по этой теме можно найти в главе 20 «Потоки выполнения».

Генераторы списков

На практике достаточно часто возникает необходимость применить некоторую функцию ко всем элементам списка, чтобы создать новый список с результатами. Например:

```
nums = [1, 2, 3, 4, 5]
squares = []
for n in nums:
    squares.append(n * n)
```

Так как потребность в подобной операции возникает очень часто, она была реализована в виде оператора, который называется *генератором списков*. Ниже приводится простой пример такого генератора:

```
nums = [1, 2, 3, 4, 5]
squares = [n * n for n in nums]
```

В общем виде генераторы списков имеют следующий синтаксис:

```
[expression for item1 in iterable1 if condition1
            for item2 in iterable2 if condition2
            ...
            for itemN in iterableN if conditionN ]
```

Этот синтаксис можно примерно выразить следующим программным кодом:

```
s = []
for item1 in iterable1:
    if condition1:
        for item2 in iterable2:
            if condition2:
                ...
                for itemN in iterableN:
                    if conditionN: s.append(expression)
```

Для иллюстрации ниже приводится несколько примеров:

```
a = [-3,5,2,-10,7,8]
b = 'abc'

c = [2*s for s in a]           # c = [-6,10,4,-20,14,16]
d = [s for s in a if s >= 0]   # d = [5,2,7,8]
e = [(x,y) for x in a          # e = [(5,'a'),(5,'b'),(5,'c'),
           for y in b          #      (2,'a'),(2,'b'),(2,'c'),
           if x > 0 ]          #      (7,'a'),(7,'b'),(7,'c'),
                               #      (8,'a'),(8,'b'),(8,'c')]

f = [(1,2), (3,4), (5,6)]
g = [math.sqrt(x*x+y*y)        # f = [2.23606, 5.0, 7.81024]
     for x,y in f]
```

Последовательности, включаемые в генераторы списков, не обязательно должны иметь одну и ту же длину, потому что итерации по их элементам выполняются с помощью вложенных циклов `for`, как было показано выше. Получающийся список содержит последовательные значения выражений. Инструкция `if` является необязательной; однако при ее использовании *выражение* вычисляется и его результат добавляется в список, только если *условие* истинно.

Если генератор списков используется для конструирования списка кортежей, значения кортежа должны заключаться в круглые скобки. Например, такой генератор списков является допустимым: `[(x,y) for x in a for y in b]`, а такой – нет: `[x,y for x in a for y in b]`.

Наконец, важно отметить, что в **Python 2 переменные циклов, определяемые** внутри генератора списков, размещаются в текущей области видимости и остаются доступными по завершении работы генератора списка. Например, переменная цикла `x`, используемая в генераторе списков `[x for x in a]`, получит значение, которое затрет значение, хранившееся в ней ранее,

и по завершении работы генератора оно будет равно значению последнего элемента полученного списка. К счастью, эта проблема была решена в Python 3, где переменная цикла остается частной.

Выражения-генераторы

Выражение-генератор – это объект, который производит те же самые вычисления, что и генератор списков, но возвращает свои результаты в процессе выполнения итераций. Выражения-генераторы имеют тот же синтаксис, что и генераторы списков, за исключением использования круглых скобок вместо квадратных. Например:

```
(expression for item1 in iterable1 if condition1
           for item2 in iterable2 if condition2
           ...
           for itemN in iterableN if conditionN)
```

В отличие от генераторов списков, выражения-генераторы не создают списки и не вычисляют немедленно выражения в скобках. Вместо этого создается объект генератора, который будет воспроизводить значения в процессе итераций. Например:

```
>>> a = [1, 2, 3, 4]
>>> b = (10*i for i in a)
>>> b
<generator object at 0x590a8>
>>> b.next()
10
>>> b.next()
20
...
```

Генераторы списков и выражения-генераторы имеют важное, но не бросающееся в глаза отличие. Генераторы списков фактически создают списки, содержащие результаты вычислений, тогда как выражения-генераторы создают объекты генераторов, которые просто могут воспроизводить данные по требованию. В некоторых случаях это может дать существенный прирост производительности и снизить объем потребляемой памяти. Например:

```
# Чтение файла
f = open("data.txt")           # Открыть файл
lines = (t.strip() for t in f) # Прочитать строки и
                               # удалить пробелы в начале и в конце
comments = (t for t in lines if t[0] == '#') # Все комментарии
for c in comments:
    print(c)
```

Выражение-генератор, которое в этом примере извлекает строки и удаляет пробельные символы в начале и в конце каждой строки, в действительности не считывает содержимое файла в память целиком. То же относится и к выражению, которое извлекает комментарии. Фактическое чтение строк из файла начинается, когда программа входит в цикл for, следую-

щий ниже. В процессе этих итераций выполняется чтение строк из файла и их фильтрация. В действительности, при таком подходе файл никогда не будет загружен в память целиком. То есть он представляет собой весьма эффективный способ извлечения комментариев из исходных файлов на языке Python огромных размеров.

В отличие от генераторов списков, выражения-генераторы не создают объекты, действующие как последовательности. К ним нельзя применить операцию индексирования или любую другую операцию, обычную для списков (например, `append()`). Однако выражение-генератор можно преобразовать в список с помощью встроенной функции `list()`:

```
clist = list(comments)
```

Декларативное программирование

Генераторы списков и выражения-генераторы неразрывно связаны с операциями, которые можно обнаружить в декларативных языках программирования. По сути эти две особенности выводятся (нестрого) из математической теории множеств. Например, выражение `[x*x for x in a if x > 0]` сходно с заданием множества $\{ x^2 \mid x \in a,\ x > 0 \}$.

Вместо того чтобы писать программы, которые вручную выполняют обход данных, с помощью этих декларативных возможностей можно структурировать программу, представив ее как серию операций, манипулирующих всеми данными сразу. Например, предположим, что имеется файл «portfolio.txt», содержащий информацию о ценных бумагах, имеющихся в наличии, например:

```
AA 100 32.20
IBM 50 91.10
CAT 150 83.44
MSFT 200 51.23
GE 95 40.37
MSFT 50 65.10
IBM 100 70.44
```

Ниже приводится программа, написанная в декларативном стиле, которая вычисляет общую стоимость, суммируя произведения значений во втором и в третьем столбцах:

```
lines  = open("portfolio.txt")
fields = (line.split() for line in lines)
print(sum(float(f[1]) * float(f[2]) for f in fields))
```

В этой программе отсутствует цикл построчного чтения содержимого файла. Вместо этого здесь просто объявляется последовательность операций, которые должны быть выполнены над всеми данными. При таком подходе мы получаем очень компактный программный код, который, к тому же, выполняется быстрее, чем более традиционная версия:

```
total = 0
for line in open("portfolio.txt"):
    fields = line.split()
```

```
    total += float(fields[1]) * float(fields[2])
print(total)
```

Декларативный стиль программирования отчасти напоминает операции, которые можно выполнять в командной оболочке UNIX. Например, предыдущий пример, использующий выражения-генераторы, можно записать в виде однострочной команды `awk`:

```
% awk '{ total += $2 * $3} END { print total }' portfolio.txt
44671.2
%
```

Кроме того, декларативный стиль генераторов списков и выражений-генераторов можно использовать для имитации поведения инструкции `select` в языке SQL, часто применяемой при работе с базами данных. Например, рассмотрим следующий пример, обрабатывающий данные, получаемые из списка словарей:

```
fields = (line.split() for line in open("portfolio.txt"))
portfolio = [ {'name' : f[0],
               'shares' : int(f[1]),
               'price' : float(f[2]) }
              for f in fields]

# Несколько запросов
msft = [s for s in portfolio if s['name'] == 'MSFT']
large_holdings = [s for s in portfolio
                        if s['shares']*s['price'] >= 10000]
```

На практике при работе с модулями, реализующими доступ к базе данных (глава 17), часто бывает удобно объединять генераторы списков и запросы к базе данных в единое целое. Например:

```
sum(shares*cost for shares,cost in
         cursor.execute("select shares, cost from portfolio")
              if shares*cost >= 10000)
```

Оператор lambda

С помощью инструкции `lambda` можно создавать анонимные функции, имеющие форму выражения:

```
lambda args : expression
```

args – это список аргументов, разделенных запятыми, а *expression* – выражение, использующее эти аргументы. Например:

```
a = lambda x,y : x+y
r = a(2,3)              # r получит значение 5
```

Программный код в инструкции `lambda` должен быть допустимым выражением. Внутри инструкции `lambda` нельзя использовать несколько инструкций или использовать другие инструкции, не являющиеся выражениями, такие как `for` или `while`. `lambda`-выражения подчиняются тем же правилам области видимости, что и функции.

Основное назначение lambda-выражений состоит в том, чтобы обеспечить возможность объявления коротких функций обратного вызова. Например, сортировку списка имен без учета регистра символов можно было бы реализовать так:

```
names.sort(key=lambda n: n.lower())
```

Рекурсия

Рекурсивные вызовы функций оформляются очень просто. Например:

```
def factorial(n):
    if n <= 1: return 1
    else: return n * factorial(n - 1)
```

Однако следует помнить, что существует ограничение на глубину рекурсии. Текущее максимальное значение количества рекурсивных вызовов можно получить с помощью функции sys.getrecursionlimit(), а изменить – с помощью функции sys.setrecursionlimit(). По умолчанию этот предел равен 1000. Несмотря на существующую возможность изменить это значение, глубина рекурсии в программах по-прежнему ограничена размером стека, который устанавливается операционной системой. По достижении максимальной глубины рекурсии возбуждается исключение RuntimeError. Интерпретатор Python не предусматривает оптимизацию хвостовой рекурсии, которая зачастую поддерживается функциональными языками программирования, такими как Scheme.

Как и следовало бы ожидать, рекурсия не может использоваться в функциях-генераторах и в сопрограммах. Например, следующий фрагмент выводит все элементы многоуровневой коллекции списков:

```
def flatten(lists):
    for s in lists:
        if isinstance(s,list):
            flatten(s)
        else:
            print(s)

items = [[1,2,3],[4,5,[5,6]],[7,8,9]]
flatten(items)     # Выведет 1 2 3 4 5 6 7 8 9
```

Но если заменить оператор print инструкцией yield, эта функция перестанет работать. Это обусловлено тем, что рекурсивный вызов flatten() просто приведет к созданию еще одного объекта генератора, по которому фактически не будет выполнено ни одной итерации. Ниже приводится действующая рекурсивная версия генератора:

```
def genflatten(lists):
    for s in lists:
        if isinstance(s,list):
            for item in genflatten(s):
                yield item
        else:
            yield item
```

Нужно внимательно отслеживать использование совместно рекурсивных функций и декораторов. Если применить декоратор к рекурсивной функции, все внутренние рекурсивные вызовы будут обращаться к декорированной версии. Например:

```
@locked
def factorial(n):
    if n <= 1: return 1
    else: return n * factorial(n - 1) # Вызов обернутой версии factorial
```

Если декоратор имеет какое-либо отношение к управлению системными ресурсами, такими как механизмы синхронизации или блокировки, от рекурсии лучше воздержаться.

Строки документирования

На практике часто можно увидеть на месте первой инструкции функции строку документирования, описывающую порядок использования этой функции. Например:

```
def factorial(n):
    """Вычисляет факториал числа n. Например:

        >>> factorial(6)
        120
        >>>
    """
    if n <= 1: return 1
    else: return n*factorial(n-1)
```

Строка документирования сохраняется в атрибуте `__doc__` функции, который часто используется интегрированными средами разработки для предоставления интерактивной справки.

Однако обертывание функций с помощью декораторов может препятствовать работе справочной системы, связанной со строками документирования. Рассмотрим в качестве примера следующий фрагмент:

```
def wrap(func):
    call(*args,**kwargs):
        return func(*args,**kwargs)
    return call
@wrap
def factorial(n):
    """Вычисляет факториал числа n."""
    ...
```

Если теперь запросить справочную информацию для этой версии функции `factorial()`, интерпретатор вернет довольно странное пояснение:

```
>>> help(factorial)
Help on function call in module __main__:
(Справка для функции call в модуле __main__:)
```

```
call(*args, **kwargs)
(END)
>>>
```

Чтобы исправить эту проблему, декоратор функций должен скопировать имя и строку документирования оригинальной функции в соответствующие атрибуты декорированной версии. Например:

```
def wrap(func):
    call(*args,**kwargs):
        return func(*args,**kwargs)
    call.__doc__ = func.__doc__
    call.__name__ = func.__name__
    return call
```

Для решения этой достаточно типичной проблемы в модуле `functools` имеется функция `wraps`, которая автоматически копирует эти атрибуты. Неудивительно, что она тоже является декоратором:

```
from functools import wraps
def wrap(func):
    @wraps(func)
    call(*args,**kwargs):
        return func(*args,**kwargs)
    return call
```

Декоратор `@wraps(`*`func`*`)`, объявленный в модуле `functools`, копирует атрибуты функции `func` в атрибуты обернутой версии функции.

Атрибуты функций

Функции допускают возможность присоединения к ним любых атрибутов. Например:

```
def foo():
    инструкции

foo.secure = 1
foo.private = 1
```

Атрибуты функции сохраняются в словаре, доступном в виде атрибута `__dict__` функции.

В основном атрибуты функций используются в узкоспециализированных приложениях, таких как генераторы парсеров и прикладные фреймворки, которые часто пользуются возможностью присоединения дополнительной информации к объектам функций.

Как и в случае со строками документирования, атрибуты функций требуют особого внимания при использовании декораторов. Если некоторая функция обернута декоратором, попытки обращения к ее атрибутам фактически будут переадресовываться к декорированной версии. Это может быть как желательно, так и нежелательно, в зависимости от потребностей приложения. Чтобы скопировать уже определенные атрибуты функции

в атрибуты декорированной версии, можно использовать приведенный далее подход или воспользоваться декоратором `functools.wraps()`, как было показано в предыдущем разделе:

```
def wrap(func):
    call(*args,**kwargs):
        return func(*args,**kwargs)
    call.__doc__ = func.__doc__
    call.__name__ = func.__name__
    call.__dict__.update(func.__dict__)
    return call
```

Функции eval(), exec() и compile()

Функция `eval(`*str* `[,`*globals* `[,`*locals*`]])` выполняет выражение в строке *str* и возвращает результат. Например:

```
a = eval('3*math.sin(3.5+x) + 7.2')
```

Аналогично функция `exec(`*str* `[, `*globals* `[, `*locals*`]])` выполняет строку *str*, содержащую произвольный программный код на языке Python. Этот программный код выполняется так, как если бы он фактически находился на месте `exec()`. Например:

```
a = [3, 5, 10, 13]
exec("for i in a: print(i)")
```

Следует отметить, что в Python 2 функция `exec()` определена как инструкция. То есть в устаревшем программном коде можно увидеть обращения к инструкции `exec`, в которой отсутствуют круглые скобки, например: `exec "for i in a: print i"`. Такой способ по-прежнему можно использовать в версии Python 2.6, но он считается недопустимым в Python 3. В новых программах эта инструкция должна вызываться как функция `exec()`.

Обе эти функции выполняют программный код в пространстве имен вызывающего программного кода (которое используется для разрешения любых имен, появляющихся в строке или в файле). Кроме того, функции `eval()` и `exec()` могут принимать один или два необязательных объекта отображений, которые для выполняемого программного кода будут играть роль глобального и локального пространств имен соответственно. Например:

```
globals = {'x': 7,
           'y': 10,
           'birds': ['Parrot', 'Swallow', 'Albatross']
           }
locals = { }

# Словари, объявленные выше, используются, как глобальное и
# локальное пространства имен при выполнении следующей инструкции
a = eval("3 * x + 4 * y", globals, locals)
exec("for b in birds: print(b)", globals, locals)
```

Если опустить одно из этих пространств имен, будут использоваться текущие глобальное или локальное пространство имен. Кроме того, из-за про-

блем, связанных с возможностью вложения областей видимости, вызов `exec()` внутри некоторой функции может приводить к исключению `SyntaxError`, если эта функция содержит вложенные определения функций или `lambda`-операторы.

При передаче строки функции `exec()` или `eval()` интерпретатор сначала скомпилирует ее в байт-код. Это делает весь процесс достаточно дорогостоящим делом, поэтому если программный код предполагается использовать неоднократно, его лучше сначала скомпилировать, а затем пользоваться скомпилированной версией.

Функция `compile(`*`str`*`,`*`filename`*`,`*`kind`*`)` компилирует строку в байт-код, где *`str`* – это строка с программным кодом, подлежащим компиляции, а *`filename`* – файл, в котором эта строка определена (для использования в сообщениях с трассировочной информацией). Аргумент *`kind`* определяет тип компилируемого программного кода: `'single'` – для единственной инструкции, `'exec'` – для множества инструкций и `'eval'` – для выражений. Объект с программным кодом, возвращаемый функцией `compile()`, может передаваться функции `eval()` и инструкции `exec()`. Например:

```
s = "for i in range(0,10): print(i)"
c = compile(s,'','exec')      # Скомпилировать в объект с программным кодом
exec(c)                       # Выполнить

s2 = "3 * x + 4 * y"
c2 = compile(s2, '', 'eval')  # Скомпилировать в выражение
result = eval(c2)             # Выполнить
```

7

Классы и объектно-ориентированное программирование

Классы – это механизм создания новых типов объектов. Эта глава подробно рассказывает о классах, но в ней не следует видеть всеохватывающее руководство по объектно-ориентированному программированию и проектированию. Здесь предполагается, что читатель уже имеет некоторый опыт работы со структурами данных и владеет приемами объектно-ориентированного программирования в других языках программирования, таких как C или Java. (Дополнительные сведения о терминологии и внутренней реализации объектов приводятся в главе 3 «Типы данных и объекты».)

Инструкция class

Класс определяет набор атрибутов, ассоциированных с ним, и используемых коллекцией объектов, которые называются *экземплярами*. Обычно класс – это коллекция функций (известных, как *методы*), переменных (известных, как *переменные класса*) и вычисляемых атрибутов (известных, как *свойства*).

Класс объявляется с помощью инструкции `class`. Тело класса составляет последовательность инструкций, которые выполняются на этапе определения класса. Например:

```
class Account(object):
    num_accounts = 0
    def __init__(self,name,balance):
        self.name = name
        self.balance = balance
        Account.num_accounts += 1
    def __del__(self):
        Account.num_accounts -= 1
    def deposit(self,amt):
        self.balance = self.balance + amt
```

```
    def withdraw(self,amt):
        self.balance = self.balance - amt
    def inquiry(self):
        return self.balance
```

Значения, создаваемые при выполнении тела класса, помещаются в объект класса, который играет роль пространства имен, во многом подобно модулю. Например, ниже показано, как осуществляется доступ к членам класса `Account`:

```
Account.num_accounts
Account.__init__
Account.__del__
Account.deposit
Account.withdraw
Account.inquiry
```

Важно отметить, что сама по себе инструкция `class` не создает никаких экземпляров класса (например, в предыдущем примере не создается никаких объектов типа `Accounts`). Вместо этого выполняется подготовка атрибутов, общих для всех экземпляров, которые будут созданы позднее. В этом смысле определение класса можно представить как шаблон.

Функции, определяемые внутри класса, называются *методами экземпляров*. Метод экземпляра – это функция, оперирующая экземпляром класса, который передается ей в первом аргументе. В соответствии с соглашениями этому аргументу дается имя `self`, хотя точно так же можно использовать любое другое имя. Функции `deposit()`, `withdraw()` и `inquiry()`, определяемые предыдущем примере, – это методы экземпляра.

Переменные класса, такие как `num_accounts`, – это значения, которые совместно используются всеми экземплярами класса (то есть они не могут содержать отдельные значения для каждого из экземпляров). В данном случае переменная хранит количество созданных экземпляров класса `Account`.

Экземпляры класса

Экземпляры класса создаются обращением к объекту класса, как к функции. В результате этого создается новый экземпляр, который затем передается методу `__init__()` класса. В число аргументов, передаваемых методу `__init__()`, входят: вновь созданный экземпляр `self` и дополнительные аргументы, указанные при вызове объекта класса. Например:

```
# Создать новый счет
a = Account("Гвидо", 1000.00) # Вызовет Account.__init__(a,"Гвидо",1000.00)
b = Account("Билл", 10.00)
```

Внутри метода `__init__()` создаются атрибуты экземпляра путем присваивания значений атрибутам объекта `self`. Например, инструкция `self.name = name` создаст атрибут `name` экземпляра. После того как вновь созданный экземпляр будет возвращен пользователю, к его атрибутам, как и к атрибутам класса, можно будет обратиться с помощью оператора точки (.), как показано ниже:

```
a.deposit(100.00) # Вызовет Account.deposit(a,100.00)
b.withdraw(50.00) # Вызовет Account.withdraw(b,50.00)
name = a.name     # Получить имя владельца счета
```

Оператор точки (.) отвечает за доступ к атрибутам. При обращении к атрибуту полученное значение может поступать из нескольких мест. Например, выражение `a.name` в предыдущем примере вернет значение атрибута `name` экземпляра `a`. Но выражение `a.deposit` вернет значение атрибута `deposit` (метод) класса `Account`. При обращении к атрибуту поиск требуемого имени сначала производится в экземпляре, и если он не увенчается успехом, поиск выполняется в классе экземпляра. Это дает классам возможность обеспечить совместное использование своих атрибутов всеми его экземплярами.

Правила видимости

Классы определяют собственные пространства имен, но они не образуют области видимости для имен, используемых внутри методов. То есть при реализации классов ссылки на атрибуты и методы должны быть полностью квалифицированы. Например, ссылки на атрибуты в методах всегда должны производиться относительно имени `self`. По этой причине для примера выше следует использовать ссылку `self.balance`, а не `balance`. То же относится и к вызовам методов из других методов, как показано в следующем примере:

```
class Foo(object):
    def bar(self):
        print("bar!")
    def spam(self):
        bar(self)  # Ошибка! Обращение к 'bar' приведет к исключению NameError
        self.bar()    # Правильно
        Foo.bar(self) # Тоже правильно
```

Отсутствие собственной области видимости в методах классов – это одна из особенностей, отличающих Python от C++ или Java. Для тех, кому раньше приходилось работать с этими языками программирования, отмечу, что аргумент `self` в языке Python – это то же самое, что указатель `this`. Необходимость явного использования `self` обусловлена тем, что язык Python не предоставляет средств явного объявления переменных, аналогичных, например, `int x` или `float y` в языке C. Без этого невозможно определить, что подразумевает операция присваивания в методе, – сохранение значения в локальной переменной или в атрибуте экземпляра. Явное использование имени `self` устраняет эту неоднозначность – все значения, сохраняемые с помощью `self`, становятся частью экземпляра, а все остальные операции присваивания действуют с локальными переменными.

Наследование

Наследование – это механизм создания новых классов, призванный настроить или изменить поведение существующего класса. Оригинальный класс называют *базовым классом* или *суперклассом*. Новый класс назы-

вают *производным классом* или *подклассом*. Когда новый класс создается с использованием механизма наследования, он «наследует» атрибуты базовых классов. Однако производный класс может переопределить любой из этих атрибутов и добавить новые атрибуты.

Наследование определяется перечислением в инструкции `class` имен базовых классов через запятые. В случае отсутствия подходящего базового класса определяется наследование класса `object`, как было показано в предыдущих примерах. `object` – это класс, который является родоначальником всех объектов в языке Python и предоставляет реализацию по умолчанию некоторых общих методов, таких как `__str__()`, создающий строковое представление объекта для вывода инструкцией `print`.

Наследование часто используется для переопределения поведения существующих методов. Например, ниже приводится специализированная версия класса `Account`, где переопределяется метод `inquiry()`, который время от времени будет возвращать значение баланса, превышающее фактическое значение, – в надежде, что невнимательный клиент превысит сумму счета и будет подвергнут большому штрафу при последующем платеже по возникшему кредиту:

```
import random
class EvilAccount(Account):
    def inquiry(self):
        if random.randint(0,4) == 1:
            return self.balance * 1.10 # Внимание: патентуем идею
        else:
            return self.balance

c = EvilAccount("Джордж", 1000.00)
c.deposit(10.0)          # Вызов Account.deposit(c,10.0)
available = c.inquiry()  # Вызов EvilAccount.inquiry(c)
```

В этом примере экземпляры класса `EvilAccount` будут идентичны экземплярам класса `Account`, за исключением переопределенного метода `inquiry()`.

Поддержка механизма наследования в языке Python реализована за счет незначительного расширения оператора точки (`.`). А именно, если поиск атрибута в экземпляре или в классе экземпляра не увенчался успехом, он продолжается в базовом классе. Этот процесс продолжается, пока не останется не просмотренных базовых классов. Это объясняет, почему вызов `c.deposit()` в предыдущем примере приводит к вызову метода `deposit()`, объявленного в классе `Account`.

Подкласс может добавлять к экземплярам новые атрибуты, определяя собственную версию метода `__init__()`. Например, следующая версия класса `EvilAccount` добавляет новый атрибут `evilfactor`:

```
class EvilAccount(Account):
    def __init__(self,name,balance,evilfactor):
        Account.__init__(self,name,balance) # Вызов метода инициализации
                                            # базового класса Account
        self.evilfactor = evilfactor
    def inquiry(self):
        if random.randint(0,4) == 1:
```

```
            return self.balance * self.evilfactor
        else:
            return self.balance
```

Когда производный класс определяет собственный метод `__init__()`, методы `__init__()` базовых классов перестают вызываться автоматически. Поэтому производный класс должен следить за выполнением инициализации базовых классов, вызывая их методы `__init__()`. В предыдущем примере этот прием можно наблюдать в строке, где вызывается метод `Account.__init__()`. Если базовый класс не имеет своего метода `__init__()`, этот шаг может быть пропущен. Если заранее не известно, определяется ли в базовом классе метод `__init__()`, для надежности можно вызвать его без аргументов, потому что в конечном итоге всегда существует реализация по умолчанию, которая просто ничего не делает.

Иногда в производном классе требуется переопределить метод, но при этом желательно вызвать оригинальную реализацию. В этом случае можно явно вызвать оригинальный метод базового класса, передав ему экземпляр `self` в первом аргументе, как показано ниже:

```
class MoreEvilAccount(EvilAccount):
    def deposit(self,amount):
        self.withdraw(5.00)              # Вычесть плату за "удобство"
        EvilAccount.deposit(self,amount) # А теперь пополнить счет
```

Главная хитрость в этом примере состоит в том, что класс `EvilAccount` в действительности не реализует метод `deposit()`. Вместо него вызывается метод класса `Account`. И хотя этот программный код работает, у тех, кто будет читать его, могут возникнуть вопросы (например, обязан ли был класс `EvilAccount` реализовать метод `deposit()`). Чтобы избежать подобной путаницы, можно использовать функцию `super()`, как показано ниже:

```
class MoreEvilAccount(EvilAccount):
    def deposit(self,amount):
        self.withdraw(5.00)                          # Вычесть плату за удобство
        super(MoreEvilAccount,self).deposit(amount)# А теперь пополнить счет
```

Функция `super(`*`cls, instance`*`)` возвращает специальный объект, позволяющий выполнять поиск атрибутов в базовых классах. При использовании этой функции интерпретатор будет искать атрибуты, следуя обычным правилам поиска в базовых классах. Это позволяет избежать жесткой привязки к имени базового класса и более ясно говорит о намерениях (то есть вы готовы вызвать предыдущую реализацию метода, независимо от того, в каком базовом классе она находится). К сожалению, синтаксис функции `super()` оставляет желать лучшего. В Python 3 можно использовать упрощенную инструкцию `super().deposit(amount)`, чтобы выполнить необходимые вычисления, показанные в примере. Однако в Python 2 необходимо использовать более многословную версию.

В языке Python поддерживается множественное наследование. Это достигается за счет указания нескольких базовых классов. Рассмотрим следующую коллекцию классов:

```
class DepositCharge(object):
    fee = 5.00
    def deposit_fee(self):
        self.withdraw(self.fee)

class WithdrawCharge(object):
    fee = 2.50
    def withdraw_fee(self):
        self.withdraw(self.fee)

# Класс, использующий механизм множественного наследования
class MostEvilAccount(EvilAccount, DepositCharge, WithdrawCharge):
    def deposit(self,amt):
        self.deposit_fee()
        super(MostEvilAccount,self).deposit(amt)
    def withdraw(self,amt):
        self.withdraw_fee()
        super(MostEvilAcount,self).withdraw(amt)
```

При использовании множественного наследования порядок поиска атрибутов становится более сложным, потому что появляется несколько возможных путей поиска. Следующие инструкции иллюстрируют эту сложность:

```
d = MostEvilAccount("Dave",500.00,1.10)
d.deposit_fee()  # Вызовет DepositCharge.deposit_fee().   fee == 5.00
d.withdraw_fee() # Вызовет WithdrawCharge.withdraw_fee(). fee == 5.00 ??
```

В этом примере методы `deposit_fee()` и `withdraw_fee()` имеют уникальные имена и обнаруживаются в соответствующих базовых классах. Однако создается ощущение, что метод `withdraw_fee()` работает с ошибкой, потому что он не использует значение атрибута `fee`, инициализированного в его классе. Это обусловлено тем, что атрибут `fee` – это переменная класса, объявленная в двух различных базовых классах. В работе используется одно из этих значений, но какое? (Подсказка: это значение атрибута `DepositCharge.fee`.)

Чтобы обеспечить поиск атрибутов при множественном наследовании, все базовые классы включаются в список, в порядке от «более специализированных» к «менее специализированным». Затем, когда производится поиск, интерпретатор просматривает этот список, пока не найдет первое определение атрибута.

В примере выше класс `EvilAccount` является более специализированным, чем класс `Account`, потому что он наследует класс `Account`. То же относится и к классу `MostEvilAccount`, `DepositCharge` считается более специализированным, чем `WithdrawCharge`, потому что он стоит первым в списке базовых классов. Порядок поиска в базовых классах можно увидеть, если вывести содержимое атрибута `__mro__` класса. Например:

```
>>> MostEvilAccount.__mro__
(<class '__main__.MostEvilAccount'>,
 <class '__main__.EvilAccount'>,
 <class '__main__.Account'>,
 <class '__main__.DepositCharge'>,
 <class '__main__.WithdrawCharge'>,
```

```
 <type 'object'>)
>>>
```

В большинстве случаев правила составления этого списка «интуитивно понятны». То есть производный класс всегда проверяется раньше его базовых классов, а если класс имеет несколько родителей, они всегда будут проверяться в том порядке, в каком были перечислены в объявлении класса. Однако точный порядок просмотра базовых классов в действительности намного сложнее и его нельзя описать с помощью какого-либо «простого» алгоритма, такого как поиск «снизу-вверх» или «слева-направо». Для упорядочения используется алгоритм С3-линеаризации, описанный в документе **«A Monotonic Superclass Linearization for Dylan» (Монотонная линеаризация суперкласса для языка Dylan)** (авторы К. Баррет (K. Barrett) и другие, был представлен на конференции **OOPSLA'96). Одна из малозаметных** особенностей этого алгоритма состоит в том, что его реализация в языке Python препятствует созданию определенных иерархий классов с возбуждением исключения `TypeError`. Например:

```
class X(object): pass
class Y(X): pass
class Z(X,Y): pass # TypeError.
                   # Невозможно определить непротиворечивый порядок
                   # разрешения имен методов
```

В данном случае алгоритм разрешения имен методов препятствует созданию `Z`, потому что он не может определить осмысленный порядок поиска в базовых классах. Например, класс `X` находится в списке родительских классов перед классом `Y`, поэтому он должен просматриваться первым. Однако класс `Y` является более специализированным, потому что наследует класс `X`. Поэтому если первым будет проверяться класс `X`, это не позволит отыскать специализированные реализации методов в классе `Y`, унаследованных от класса `X`. На практике такие ситуации должны возникать очень редко, и если возникают, это обычно свидетельствует о более серьезных ошибках, допущенных при проектировании программы.

Как правило, в большинстве программ лучше стараться избегать множественного наследования. Однако иногда множественное наследование используется для объявления так называемых *классов-примесей*. Класс-примесь, обычно определяющий набор методов, объявляется, чтобы потом «подмешивать» его в другие классы, с целью расширения их функциональных возможностей (почти как макроопределение). Обычно предполагается, что существуют другие методы, и методы в классах-примесях встраиваются поверх них. Эту возможность иллюстрируют классы `DepositCharge` и `WithdrawCharge`, объявленные в предыдущем примере. Эти классы добавляют новые методы, такие как `deposit_fee()`, в классы, включающие их в число базовых классов. Однако вам едва ли потребовалось бы, например, создавать экземпляры самого класса `DepositCharge`. В действительности экземпляр этого класса едва ли мог бы иметь практическую пользу (он обладает всего одним методом, который сам по себе не может даже работать правильно).

И еще одно последнее замечание: если в этом примере потребуется исправить проблему со ссылкой на атрибут `fee`, методы `deposit_fee()` и `withdraw_fee()` можно изменить так, чтобы они обращались к атрибуту напрямую, используя имя класса вместо ссылки `self` (например, `DepositChange.fee`).

Полиморфизм, или динамическое связывание и динамическая типизация

Динамическое связывание (иногда, в контексте наследования, называется *полиморфизм*) – это возможность использования экземпляра без учета его фактического типа. Данная возможность целиком обеспечивается механизмом поиска атрибутов в дереве наследования, описанным в предыдущем разделе. Всякий раз, когда производится обращение к атрибуту, такое как `obj.attr`, поиск атрибута `attr` сначала выполняется в самом экземпляре, затем в определении класса экземпляра, а затем в базовых классах. Поиск прекращается, как только будет найден первый атрибут с требуемым именем.

Важной особенностью процесса связывания является его независимость от того, какому типу принадлежит объект *obj*. То есть при обращении к атрибуту *obj.name* этот механизм будет работать с любым объектом *obj*, имеющим атрибут *name*. Такое поведение иногда называют *динамической типизацией* (или утиной типизацией исходя из пословицы: «если это выглядит, крякает и ходит, как утка, значит, это утка»).

Часто программисты пишут программы на языке Python, исходя из этого поведения. Например, когда необходимо создать модифицированную версию существующего класса, на его основе можно либо создать производный класс либо определить совершенно новый класс, который выглядит и действует как прежний, но никак с ним не связанный. Последний подход часто используется для ослабления связей между компонентами программы. Например, можно написать программный код, который будет работать с объектами любых типов, при условии, что они обладают определенным набором методов. В качестве одного из наиболее типичных примеров можно привести различные объекты, напоминающие файлы, которые определяются в стандартной библиотеке. Хотя эти объекты своим поведением напоминают файлы, тем не менее они не наследуют встроенный объект файла.

Статические методы и методы классов

По умолчанию предполагается, что все функции, присутствующие в определении класса, будут оперировать экземпляром, который всегда передается в виде первого аргумента `self`. Однако существуют еще два типа методов, которые можно определить.

Статический метод – это обычная функция, которая просто включается в пространство имен, определяемое классом. Она не оперирует какими-либо экземплярами. Для определения статических методов используется декоратор `@staticmethod`, как показано ниже:

```
class Foo(object):
    @staticmethod
    def add(x,y):
        return x + y
```

Чтобы вызвать статический метод, достаточно просто добавить имя класса перед ним. При этом не требуется передавать ему какую-либо дополнительную информацию. Например:

```
x = Foo.add(3,4) # x = 7
```

Обычно статические методы используются для обеспечения различных способов создания новых экземпляров. В объявлении класса может быть только один метод `__init__()`, однако имеется возможность объявить альтернативные функции создания экземпляров, как показано ниже:

```
class Date(object):
    def __init__(self,year,month,day):
        self.year = year
        self.month = month
        self.day = day
    @staticmethod
    def now():
        t = time.localtime()
        return Date(t.tm_year, t.tm_mon, t.tm_day)
    @staticmethod
    def tomorrow():
        t = time.localtime(time.time()+86400)
        return Date(t.tm_year, t.tm_mon, t.tm_day)

# Несколько примеров создания экземпляров
a = Date(1967, 4, 9)
b = Date.now()      # Вызовет статический метод now()
c = Date.tomorrow() # Вызовет статический метод tomorrow()
```

Методы класса – это методы, которые оперируют самим классом как объектом. Определяются они с помощью декоратора `@classmethod`. Метод класса отличается от метода экземпляра тем, что в первом аргументе, который в соответствии с соглашениями называется *`cls`*, ему передается класс. Например:

```
class Times(object):
    factor = 1
    @classmethod
    def mul(cls,x):
        return cls.factor*x

class TwoTimes(Times):
    factor = 2

x = TwoTimes.mul(4) # Вызовет Times.mul(TwoTimes, 4) -> 8
```

Обратите внимание, что в данном примере класс `TwoTimes` передается методу `mul()` как объект. Этот пример во многом искусственный, тем не менее существуют практические, и весьма тонкие, применения методов классов.

Например, предположим, что объявляется класс, наследующий класс Date, показанный выше, и немного модифицирующий его:

```
class EuroDate(Date):
    # Изменена строка преобразования, чтобы обеспечить возможность
    # представления дат в европейском формате
    def __str__(self):
        return "%02d/%02d/%4d" % (self.day, self.month, self.year)
```

Поскольку этот класс наследует класс Date, он обладает всеми его особенностями. Однако поведение методов now() и tomorrow() будет немного портить общую картину. Например, если вызвать метод EuroDate.now(), вместо объекта EuroDate он вернет объект Date. Это можно исправить, изменив метод класса:

```
class Date(object):
    ...
    @classmethod
    def now(cls):
        t = time.localtime()
        # Создать объект соответствующего типа
        return cls(t.tm_year, t.tm_month, t.tm_day)

class EuroDate(Date):
    ...

a = Date.now()     # Вызовет Date.now(Date) и вернет Date
b = EuroDate.now() # Вызовет Date.now(EuroDate) и вернет EuroDate
```

Одна из особенностей статических методов и методов класса состоит в том, что эти методы располагаются в том же пространстве имен, что и методы экземпляра. Вследствие этого они могут вызываться относительно экземпляра. Например:

```
a = Date(1967,4,9)
b = a.now()        # Вызовет Date.now(Date)
```

Это может быть источником недопонимания, потому что вызов a.now() в действительности никак не воздействует на экземпляр a. Такое поведение является одной из особенностей объектной системы языка Python, которые отличают его от других объектно-ориентированных языков программирования, таких как Smalltalk и Ruby. В этих языках методы класса отделены от методов экземпляра.

Свойства

Обычно при обращении к атрибуту экземпляра или класса возвращается значение, сохраненное в этом атрибуте ранее. *Свойство* – это особая разновидность атрибута, который вычисляет свое значение при попытке обращения к нему. Ниже приводится простой пример:

```
class Circle(object):
    def __init__(self,radius):
        self.radius = radius
    # Некоторые дополнительные свойства класса Circles
```

```
    @property
    def area(self):
        return math.pi*self.radius**2
    @property
    def perimeter(self):
        return 2*math.pi*self.radius
```

Благодаря этому получившийся объект `Circle` обрел следующие особенности:

```
>>> c = Circle(4.0)
>>> c.radius
4.0
>>> c.area
50.26548245743669
>>> c.perimeter
25.132741228718345
>>> c.area = 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  AttributeError: can't set attribute
(Перевод:
Трассировочная информация (самый последний вызов - самый нижний):
  Файл "<stdin>", строка 1, в <module>
AttributeError: невозможно установить значение атрибута
)
>>>
```

В этом примере экземпляры класса `Circle` обладают переменной экземпляра `c.radius`, где хранится значение, и атрибутами `c.area` и `c.perimeter`, значения которых вычисляются исходя из значения этой переменной. Декоратор `@property` обеспечивает возможность обращения к методу, следующему за ним, как к простому атрибуту, без круглых скобок `()`, которые обычно добавляются, чтобы вызвать метод. Объект не имеет никаких отличительных признаков, которые говорили бы о том, что значение атрибута вычисляется, – кроме вывода сообщения об ошибке, которое генерируется при попытке переопределить значение атрибута (о чем свидетельствует исключение `AttributeError` в примере выше).

Такой способ использования свойств имеет прямое отношение к реализации *принципа единообразного доступа* (*Uniform Access Principle*). Суть состоит в том, что когда объявляется класс, хорошо бы обеспечить максимальное единообразие доступа к нему. Без применения свойств доступ к одним атрибутам выглядел бы, как обращение к обычным атрибутам, например `c.radius`, а к другим – как к методам, например `c.area()`. Необходимость запоминать, когда следует добавлять круглые скобки `()`, а когда – нет, лишь вносит лишнюю путаницу. Свойства помогают избавиться от этих неприятностей.

Программисты на языке Python не всегда понимают, что сами методы неявно интерпретируются, как свойства. Рассмотрим следующий класс:

```
class Foo(object):
    def __init__(self,name):
```

```
        self.name = name
    def spam(self,x):
        print("%s, %s" % (self.name, x)
```

Когда пользователь создаст экземпляр этого класса, например: `f = Foo ("Гвидо")`, и попробует обратиться к атрибуту `f.spam`, он получит не объект функции `spam`, а то, что называется *связанным методом*, то есть объект, представляющий вызов метода, который будет выполнен при добавлении к нему оператора вызова `()`. Связанный метод напоминает частично подготовленную функцию, для которой аргумент `self` уже имеет некоторое значение, но которой еще необходимо передать дополнительные аргументы при вызове с помощью оператора `()`. Создание этого связанного метода производится функцией свойства, которая вызывается за кулисами. Когда с помощью декораторов `@staticmethod` и `@classmethod` создается статический метод или метод класса, фактически выбирается другая функция свойства, которая обеспечит немного иной способ обращения к этим методам. Например, декоратор `@staticmethod` просто возвращает функцию метода в том же виде, в каком получил ее, ничего не добавляя и не изменяя.

Свойства также могут перехватывать операции по изменению и удалению атрибута. Делается это посредством присоединения к свойству специальных методов изменения и удаления. Например:

```
class Foo(object):
    def __init__(self,name):
        self.__name = name
    @property
    def name(self):
        return self.__name
    @name.setter
    def name(self,value):
        if not isinstance(value,str):
            raise TypeError("Имя должно быть строкой!")
        self.__name = value
    @name.deleter
    def name(self):
        raise TypeError("Невозможно удалить атрибут name")

f = Foo("Гвидо")
n = f.name       # вызовет f.name() - вернет функцию
f.name = "Монти" # вызовет метод изменения name(f,"Монти")
f.name = 45      # вызовет метод изменения name(f,45) -> TypeError
del f.name       # вызовет метод удаления name(f) -> TypeError
```

Сначала в этом примере с помощью декоратора `@property` и ассоциированного с ним метода объявляется атрибут `name` как свойство, доступное только для чтения. Следующие ниже декораторы `@name.setter` и `@name.deleter` связывают дополнительные методы с операциями изменения и удаления атрибута `name`. Имена этих методов должны в точности совпадать с именем оригинального свойства. Обратите внимание, что в этих методах фактическое значение свойства `name` сохраняется в атрибуте `__name`. Имя этого атрибута не должно следовать каким-либо соглашениям, но оно должно отличаться от имени свойства, чтобы избежать неоднозначности.

В старом программном коде часто можно встретить определения свойств, выполненные с помощью функции property(*getf*=None, *setf*=None, *delf*=None, *doc*=None), которой передаются методы с уникальными именами, реализующие необходимые операции. Например:

```
class Foo(object):
    def getname(self):
        return self.__name
    def setname(self,value):
        if not isinstance(value,str):
            raise TypeError("Имя должно быть строкой!")
        self.__name = value
    def delname(self):
        raise TypeError("Невозможно удалить атрибут name")
    name = property(getname,setname,delname)
```

Этот устаревший подход по-прежнему поддерживается, но использование декораторов позволяет получать более удобные определения классов. Например, при использовании декораторов функции get, set и delete не будут видны как методы.

Дескрипторы

При использовании свойств доступ к атрибутам управляется серией пользовательских функций get, set и delete. Такой способ управления атрибутами может быть обобщен еще больше, за счет использования *объекта дескриптора*. Дескриптор – **это обычный объект, представляющий значение** атрибута. За счет реализации одного или более специальных методов __get__(), __set__() и __delete__() он может подменять механизмы доступа к атрибутам и влиять на выполнение этих операций. Например:

```
class TypedProperty(object):
    def __init__(self,name,type,default=None):
        self.name = "_" + name
        self.type = type
        self.default = default if default else type()
    def __get__(self,instance,cls):
        return getattr(instance,self.name,self.default)
    def __set__(self,instance,value):
        if not isinstance(value,self.type):
            raise TypeError("Значение должно быть типа %s" % self.type)
        setattr(instance,self.name,value)
    def __delete__(self,instance):
        raise AttributeError("Невозможно удалить атрибут")

class Foo(object):
    name = TypedProperty("name",str)
    num = TypedProperty("num",int,42)
```

В этом примере класс TypedProperty определяет дескриптор, выполняющий проверку типа при присваивании значения атрибуту и возбуждающий исключение при попытке удалить атрибут. Например:

```
f = Foo()
a = f.name        # Неявно вызовет Foo.name.__get__(f,Foo)
f.name = "Гвидо"  # Вызовет Foo.name.__set__(f,"Guido")
del f.name        # Вызовет Foo.name.__delete__(f)
```

Экземпляры дескрипторов могут создаваться только на уровне класса. Нельзя создавать объекты дескрипторов для каждого экземпляра в отдельности, внутри метода `__init__()` или в других методах. Кроме того, имя атрибута, используемое для сохранения дескриптора в классе, имеет более высокий приоритет перед другими атрибутами на уровне экземпляров. Именно поэтому в предыдущем примере объекту дескриптора передается параметр `name` с именем, и именно поэтому полученное имя изменяется за счет добавления ведущего символа подчеркивания. Чтобы дескриптор мог сохранять значение атрибута в экземпляре, имя этого атрибута должно отличаться от имени, используемого самим дескриптором.

Инкапсуляция данных и частные атрибуты

По умолчанию все атрибуты и методы класса являются общедоступными. Это означает, что все они доступны без каких-либо ограничений. Это также означает, что все атрибуты и методы базового класса будут унаследованы и доступны в производных классах. В объектно-ориентированном программировании эта особенность часто бывает нежелательной, потому что позволяет легко получить информацию о внутреннем устройстве объекта и может привести к конфликту имен между объектами, созданными на основе базового и производного классов.

Чтобы исправить эту проблему, все имена в определении класса, начинающиеся с двух символов подчеркивания, такие как `__Foo`, автоматически изменяются и обретают вид *`_Classname__Foo`*. Благодаря этому обеспечивается эффективный способ создания частных атрибутов и методов класса, потому что частные имена в порожденном классе не будут конфликтовать с такими же частными именами в базовом классе. Например:

```
class A(object):
    def __init__(self):
        self.__X = 3     # Будет изменено на self._A__X
    def __spam(self):    # Будет изменено на _A__spam()
        pass
    def bar(self):
        self.__spam()    # Вызовет только метод A.__spam()

class B(A):
    def __init__(self):
        A.__init__(self)
        self.__X = 37    # Будет изменено на self._B__X
    def __spam(self):    # Будет изменено на _B__spam()
        pass
```

Хотя такая схема именования создает иллюзию сокрытия данных, на самом деле не существует механизма, который действительно препятствовал бы попыткам обратиться к «частным» атрибутам класса. Так, если имя

класса и имя частного атрибута известно заранее, к нему можно обратиться, использовав измененное имя. Класс может сделать такие атрибуты менее заметными, переопределив метод `__dir__()`, который формирует список имен, возвращаемый функцией `dir()`, используемой для исследования объектов.

На первый взгляд, такое изменение имен выглядит, как дополнительная операция, на выполнение которой тратятся вычислительные ресурсы, но в реальности она выполняется один раз, на этапе определения класса. Она не производится в процессе выполнения методов и не влечет за собой снижение производительности программы. Кроме того, следует отметить, что подмена имен не происходит в таких функциях, как `getattr()`, `hasattr()`, `setattr()` или `delattr()`, где имена атрибутов передаются в виде строк. При работе с этими функциями для доступа к атрибутам необходимо явно указывать измененные имена атрибутов в виде *`_Classname__name`*.

Рекомендуется определять частные атрибуты с изменяемыми значениями через свойства. Тем самым вы подтолкнете пользователя к использованию имени свойства, а не самого объекта данных (что, вероятно, желательно для вас, иначе вы не стали бы оборачивать данные в свойство). Пример такого подхода приводится в следующем разделе.

Создание частных методов обеспечивает для суперкласса возможность предотвратить переопределение и изменение реализаций этих методов в порожденных классах. Например, метод `A.bar()`, в примере выше, будет вызывать только метод `A.__spam()`, независимо от типа аргумента `self` или от наличия другого метода `__spam()` в производном классе.

Наконец, не следует путать правила именования частных атрибутов класса с правилами именования «частных» определений в модуле. Типичная ошибка состоит в попытке определить частный атрибут класса с единственным символом подчеркивания в начале (например, `_name`). В модулях это соглашение об именовании позволяет препятствовать экспортированию имен с помощью инструкции `import *`. Однако в классах этот прием не приводит к сокрытию атрибутов и не предотвращает конфликты имен, которые могут возникнуть, если в производном классе будет предпринята попытка определить новый атрибут или метод с тем же именем.

Управление памятью объектов

Когда объявляется класс, в результате получается класс, который служит фабрикой по производству экземпляров новых экземпляров. Например:

```
class Circle(object):
    def __init__(self,radius):
        self.radius = radius

# Создать несколько экземпляров класса Circle
c = Circle(4.0)
d = Circle(5.0)
```

Создание экземпляра выполняется в два этапа, с использованием специального метода `__new__()`, создающего новый экземпляр, и метода `__init__()`,

инициализирующего его. Например, операция `c = Circle(4.0)` выполняется в два этапа:

```
c = Circle.__new__(Circle, 4.0)
if isinstance(c,Circle):
    Circle.__init__(c,4.0)
```

Метод `__new__()` класса достаточно редко определяется в программах. Если он определяется, то обычно его объявление следует шаблону `__new__(cls, *args, **kwargs)`, где `args` и `kwargs` – это те же самые аргументы, что передаются методу `__init__()`. Метод `__new__()` всегда является методом класса, который принимает объект класса в первом аргументе. Хотя метод `__new__()` создает экземпляр, он не вызывает метод `__init__()` автоматически.

Если вам доведется увидеть объявление метода `__new__()` в классе, обычно это может означать одно из двух. Во-первых, класс может наследовать базовый класс, экземпляры которого относятся к разряду неизменяемых объектов. Эта ситуация характерна для классов, которые наследуют встроенные неизменяемые типы, такие как целые числа, строка или кортеж, потому что метод `__new__()` является единственным методом, вызываемым перед созданием экземпляра, и единственным, где такое значение может быть изменено (в этом смысле метод `__init__()` вызывается слишком поздно). Например:

```
class Upperstr(str):
    def __new__(cls,value=""):
        return str.__new__(cls, value.upper())

u = Upperstr("hello")     # Вернет значение "HELLO"
```

Другой случай, когда может потребоваться переопределить метод `__new__()`, – при объявлении метаклассов. Подробнее о них рассказывается в конце этой главы.

После создания экземпляра управление им осуществляется за счет подсчета ссылок. Когда счетчик ссылок уменьшается до нуля, экземпляр тут же уничтожается. Когда происходит уничтожение экземпляра, интерпретатор сначала отыскивает метод `__del__()` объекта и вызывает его. На практике необходимость в определении метода `__del__()` возникает редко. Единственное исключение – когда в процессе уничтожения объекта необходимо выполнить некоторые завершающие действия, например закрыть файл, разорвать сетевое соединение или освободить другие системные ресурсы. Однако даже в таких случаях нельзя полностью полагаться на метод `__del__()`, потому что нет никаких гарантий, что этот метод будет вызван при завершении работы интерпретатора. Для этих целей лучше определить метод, такой как `close()`, который программа сможет явно использовать при завершении.

Иногда для удаления ссылки на объект в программах используется инструкция `del`. Если в этом случае счетчик ссылок уменьшается до нуля, вызывается метод `__del__()`. Однако в общем случае вызов инструкции `del` не обязательно приводит к вызову метода `__del__()`.

С уничтожением объектов связана одна малозаметная проблема, которая обусловлена тем, что не все ситуации, когда должен вызываться метод `__del__()`, распознаются циклическим сборщиком мусора (что является веской причиной не определять метод `__del__()` без серьезных на то оснований). Программисты, пришедшие из других языков программирования, в которых отсутствует механизм автоматического сбора мусора (например, C++), должны стараться уклониться от стиля программирования с неоправданным использованием метода `__del__()`. Ситуации, когда объявление метода `__del__()` может нарушить работу сборщика мусора, возникают редко, тем не менее существуют определенные программные случаи, в которых это может вызывать ошибки, особенно когда речь заходит об отношениях родитель-потомок или о графах. Например, предположим, что имеется объект, реализованный в соответствии с шаблоном проектирования «наблюдатель».

```
class Account(object):
    def __init__(self,name,balance):
        self.name = name
        self.balance = balance
        self.observers = set()
    def __del__(self):
        for ob in self.observers:
            ob.close()
        del self.observers
    def register(self,observer):
        self.observers.add(observer)
    def unregister(self,observer):
        self.observers.remove(observer)
    def notify(self):
        for ob in self.observers:
            ob.update()
    def withdraw(self,amt):
        self.balance -= amt
        self.notify()

class AccountObserver(object):
    def __init__(self, theaccount):
        self.theaccount = theaccount
        theaccount.register(self)
    def __del__(self):
        self.theaccount.unregister(self)
        del self.theaccount
    def update(self):
        print("Баланс: %0.2f" % self.theaccount.balance)
    def close(self):
        print("Наблюдение за счетом окончено")

# Пример создания
a = Account('Дейв',1000.00)
a_ob = AccountObserver(a)
```

В этом фрагменте объявляется класс Account, позволяющий устанавливать множество объектов класса AccountObserver для наблюдения за экземпляром Account, которые извещаются о любых изменениях баланса. Для этого каждый экземпляр Account сохраняет ссылки на множество объектов-наблюдателей, а каждый экземпляр AccountObserver хранит ссылку на объект счета. Каждый из этих классов объявляет метод __del__() в надежде получить возможность выполнять некоторые заключительные операции (такие как закрытие связи между объектами и так далее). Однако **это просто** не работает. Дело в том, что между объектами образуются циклические ссылки, вследствие этого счетчик ссылок никогда не достигает нуля и заключительные операции не выполняются. Но и это еще не все, сборщик мусора (модуль gc) не может даже удалить эти объекты, что приводит к постоянной утечке памяти.

Один из способов исправить эту проблему, приведенный в следующем примере, состоит в том, чтобы с помощью модуля weakref создать слабую ссылку из одного класса на другой класс. *Слабая ссылка* – это такая ссылка на объект, создание которой не приводит к увеличению счетчика ссылок. При использовании слабых ссылок необходимо добавлять дополнительный программный код, который будет проверять наличие объекта по ссылке. Ниже приводится пример измененного класса наблюдателя:

```
import weakref
class AccountObserver(object):
    def __init__(self, theaccount):
        self.accountref = weakref.ref(theaccount) # Создаст слабую ссылку
        theaccount.register(self)
    def __del__(self):
        acc = self.accountref()    # Вернет объект счета
        if acc:                    # Прекратить наблюдение, если существует
            acc.unregister(self)
    def update(self):
        print("Баланс: %0.2f" % self.accountref().balance)
    def close(self):
        print("Наблюдение за счетом окончено")

# Пример создания
a = Account('Дейв',1000.00)
a_ob = AccountObserver(a)
```

В этом примере создается слабая ссылка accountref. Чтобы получить объект Account, на который она ссылается, к ней следует обращаться, как к функции. Это вызов вернет либо объект Account, либо None, если наблюдаемый объект больше не существует. После внесения этих изменений циклические ссылки больше не создаются. Если теперь объект Account будет уничтожен, интерпретатор вызовет его метод __del__, и все наблюдатели получат извещение. Кроме того, исчезнут препятствия в работе модуля gc. Более подробная информация о модуле weakref приводится в главе 13 «Службы Python времени выполнения».

Представление объектов и связывание атрибутов

Атрибуты объектов внутри реализованы с использованием словаря, доступного в виде атрибута `__dict__`. Этот словарь содержит уникальные данные для каждого экземпляра. Например:

```
>>> a = Account('Гвидо', 1100.0)
>>> a.__dict__
{'balance': 1100.0, 'name': 'Гвидо'}
```

Новые атрибуты могут быть добавлены к экземпляру в любой момент, например:

```
a.number = 123456    # Добавит в словарь a.__dict__ атрибут 'number'
```

Любые модификации в экземпляре всегда отражаются на содержимом локального атрибута `__dict__`. Точно так же любые модификации, выполненные непосредственно в словаре `__dict__`, отразятся на содержимом атрибутов.

Экземпляры содержат ссылки на свои классы в специальном атрибуте `__class__`. Сам по себе класс также является лишь тонкой оберткой вокруг словаря, доступного в виде атрибута `__dict__`. Словарь класса – это место, где хранятся ссылки на методы. Например:

```
>>> a.__class__
<class '__main__.Account'>
>>> Account.__dict__.keys()
['__dict__', '__module__', 'inquiry', 'deposit', 'withdraw',
'__del__', 'num_accounts', '__weakref__', '__doc__', '__init__']
>>>
```

Наконец, классы хранят ссылки на свои базовые классы в специальном атрибуте `__bases__`, который является кортежем базовых классов. Эта структура составляет основу всех операций чтения, записи и удаления атрибутов объектов.

Всякий раз, когда с помощью инструкции `obj.`*`name`* `=` *`value`* выполняется запись значения в атрибут, вызывается специальный метод `obj.__setattr__("`*`name`*`", `*`value`*`)`. Когда атрибут удаляется с помощью инструкции `del obj.`*`name`*, вызывается специальный метод `obj.__delattr__("`*`name`*`")`. По умолчанию эти методы изменяют или удаляют значения из локального словаря `__dict__` экземпляра `obj`, если требуемый атрибут не является свойством или дескриптором. В противном случае выполняются операции записи или удаления, реализованные в виде функций, связанных со свойством.

При обращении к атрибуту экземпляра, такому как `obj.`*`name`*, вызывается специальный метод `obj.__getattribute__("`*`name`*`")`. Этот метод осуществляет поиск атрибута, в процессе которого выясняется, не является ли атрибут свойством, проверяется содержимое локального атрибута `__dict__`, содержимое словаря класса и при необходимости выполняется поиск в базовых классах. Если поиск не увенчался успехом, производится последняя попытка отыскать атрибут вызовом метода `__getattr__()` класса (если он

определен). Если и эта попытка оканчивается неудачей, возбуждается исключение AttributeError.

Пользовательские классы могут реализовать собственные версии функций доступа к атрибутам, если это необходимо. Например:

```
class Circle(object):
    def __init__(self,radius):
        self.radius = radius
    def __getattr__(self,name):
        if name == 'area':
            return math.pi*self.radius**2
        elif name == 'perimeter':
            return 2*math.pi*self.radius
        else:
            return object.__getattr__(self,name)
    def __setattr__(self,name,value):
        if name in ['area','perimeter']:
            raise TypeError("%s is readonly" % name)
        object.__setattr__(self,name,value)
```

Класс, реализующий собственные версии этих методов, для выполнения основной работы, вероятно, должен опираться на реализацию по умолчанию. Это обусловлено тем, что реализация по умолчанию учитывает массу дополнительных особенностей классов, таких как дескрипторы и свойства.

Вообще говоря, в классах достаточно редко возникает необходимость переопределять операции обращения к атрибутам. Единственная область, где такая потребность возникает достаточно часто, – это разработка универсальных оберток для существующих объектов. Переопределяя методы __getattr__(), __setattr__() и __delattr__(), обертка может перехватывать обращения к атрибутам и прозрачно переадресовывать эти операции в другой объект.

__slots__

Существует возможность ограничить класс определенным набором имен атрибутов, определив специальную переменную __slots__. Например:

```
class Account(object):
    __slots__ = ('name','balance')
    ...
```

Если в классе определена переменная __slots__, экземпляры такого класса смогут иметь атрибуты только с указанными именами. В противном случае будет возбуждаться исключение AttributeError. Это ограничение исключает возможность добавления новых атрибутов к существующим экземплярам и решает проблему присваивания значений атрибутам, в именах которых допущена опечатка.

В действительности переменная __slots__ задумывалась совсем не в качестве меры предосторожности. Фактически это инструмент оптимизации по объему занимаемой памяти и скорости выполнения. Экземпляры классов, где определена переменная __slots__, уже не используют словарь

для хранения данных экземпляра. Вместо него используется более компактная структура данных, в основе которой лежит массив. Применение переменной `__slots__` в программах, создающих огромное число объектов, может существенно уменьшить объем потребляемой памяти и увеличить скорость выполнения.

Следует заметить, что переменная `__slots__` по-особенному воздействует на наследование. Если в базовом классе используется переменная `__slots__`, производный класс также должен объявлять переменную `__slots__` со списком имен своих атрибутов (даже если он не добавляет новых атрибутов), чтобы иметь возможность использовать преимущества, предоставляемые этой переменной. Если этого не сделать, производный класс будет работать медленнее и занимать памяти даже больше, чем в случае, когда переменная `__slots__` не используется ни в одном из классов!

Кроме того, использование переменной `__slots__` может нарушить работоспособность программного кода, который ожидает, что экземпляры будут иметь атрибут `__dict__`. Хотя такая ситуация не характерна для прикладного программного кода, тем не менее вспомогательные библиотеки и другие инструменты поддержки объектов могут использовать словарь `__dict__` для отладки, сериализации объектов и выполнения других операций.

Наконец, наличие объявления переменной `__slots__` не требует переопределения в классе таких методов, как `__getattribute__()`, `__getattr__()` и `__setattr__()`. По умолчанию эти методы учитывают возможность наличия переменной `__slots__`. Кроме того, следует подчеркнуть, что нет никакой необходимости добавлять имена методов и свойств в переменную `__slots__`, так как они хранятся не на уровне экземпляров, а на уровне класса.

Перегрузка операторов

Пользовательские объекты можно заставить работать со всеми встроенными операторами языка Python, добавив в класс реализации специальных методов, описанных в главе 3. Например, чтобы добавить в язык Python поддержку нового типа чисел, можно было бы объявить класс со специальными методами, такими как `__add__()`, определяющими порядок работы стандартных математических операторов с экземплярами.

Следующий ниже пример демонстрирует использование этого приема, объявляя класс, который реализует комплексные числа с поддержкой некоторых стандартных математических операторов.

Примечание

Поддержка комплексных чисел уже имеется в языке Python, поэтому данный класс приводится исключительно в демонстрационных целях.

```
class Complex(object):
    def __init__(self,real,imag=0):
        self.real = float(real)
        self.imag = float(imag)
    def __repr__(self):
```

```
        return "Complex(%s,%s)" % (self.real, self.imag)
    def __str__(self):
        return "(%g+%gj)" % (self.real, self.imag)
    # self + other
    def __add__(self,other):
        return Complex(self.real + other.real, self.imag + other.imag)
    # self - other
    def __sub__(self,other):
        return Complex(self.real - other.real, self.imag - other.imag)
```

В этом примере метод `__repr__()` возвращает строку, которая может быть использована для повторного создания объекта с помощью функции `eval()` (а именно, "`Complex(`*real*`,`*imag*`)`"). Этому соглашению должны следовать все объекты по мере возможности. Метод `__str__()`, напротив, создает строку, предназначенную для форматированного вывода (именно эту строку выводит инструкция `print`).

Другие специальные методы, такие как `__add__()` и `__sub__()`, реализуют математические операции. Реализация этих операторов тесно связана с порядком следования операндов и с их типами. В предыдущем примере, где реализованы методы `__add__()` и `__sub__()`, соответствующие им математические операторы могут применяться, *только* если комплексное число стоит слева от оператора. Они не будут вызываться, если комплексное число находится справа, а слева стоит операнд, который не является объектом типа `Complex`. Например:

```
>>> c = Complex(2,3)
>>> c + 4.0
Complex(6.0,3.0)
>>> 4.0 + c
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'Complex'
(Перевод:
Трассировочная информация (самый последний вызов - самый нижний):
  Файл "<stdin>", строка 1, в <module>
TypeError: неподдерживаемые типы операндов для +: 'int' и 'Complex'
)
>>>
```

Отчасти операция `c + 4.0` выполнилась случайно. Просто все встроенные числовые типы в языке Python уже имеют атрибуты `.real` и `.imag`, поэтому они могут участвовать в вычислениях. Если бы объект не имел этих атрибутов, попытка выполнить операцию окончилась бы неудачей. Если бы потребовалось обеспечить в классе `Complex` возможность выполнения операций с объектами, не имеющими этих атрибутов, можно было бы реализовать дополнительные преобразования (которые могут зависеть от типа объекта, представляющего другой операнд).

Попытка выполнить операцию `4.0 + c` провалилась, потому что встроенный тип чисел с плавающей точкой ничего не знает о классе `Complex`. Чтобы исправить эту проблему, можно добавить в класс `Complex` методы для работы с операндами, стоящими в обратном порядке:

```
class Complex(object):
    ...
    def __radd__(self,other):
        return Complex(other.real + self.real, other.imag + self.imag)
    def __rsub__(self,other):
        return Complex(other.real - self.real, other.imag - self.img)
    ...
```

Эти методы играют роль запасного варианта. Когда операция `4.0 + c` потерпит неудачу, прежде чем возбудить исключение `TypeError`, интерпретатор попробует вызвать метод `c.__radd__(4.0)`.

В более старых версиях Python использовались различные подходы, основанные на приведении типов операндов. Например, в устаревшем программном коде на языке Python можно встретить классы, реализующие метод `__coerce__()`. Он **больше не используется в версиях Python 2.6 и Python 3**. Кроме того, вас не должны вводить в заблуждение такие специальные методы, как `__int__()`, `__float__()` или `__complex__()`. Эти методы вызываются в операциях явного преобразования типа, таких как `int(x)` или `float(x)`, но они никогда не вызываются для неявного преобразования типов в арифметических операциях. Поэтому при создании классов с операторами, допускающими использование операндов различных типов, необходимо предусматривать явное приведение типов в реализации каждого оператора.

Типы и проверка принадлежности к классу

Когда создается экземпляр класса, типом этого экземпляра становится сам класс. Для проверки принадлежности к классу используется встроенная функция `isinstance(obj,cname)`. Эта функция возвращает значение `True`, если объект *obj* принадлежит классу *cname* или любому другому классу, производному от *cname*. Например:

```
class A(object): pass
class B(A): pass
class C(object): pass

a = A()         # Экземпляр класса 'A'
b = B()         # Экземпляр класса 'B'
c = C()         # Экземпляр класса 'C'

type(a)         # Вернет класс объекта: A
isinstance(a,A) # Вернет True
isinstance(b,A) # Вернет True, класс B - производный от класса A
isinstance(b,C) # Вернет False, класс C не является производным от класса A
```

Аналогично встроенная функция `issubclass(A,B)` вернет `True`, если класс *A* является подклассом класса *B*. Например:

```
issubclass(B,A) # Вернет True
issubclass(C,A) # Вернет False
```

Одна из проблем, связанных с проверкой типов объектов, состоит в том, что программисты часто не используют механизм наследования, а просто соз-

дают объекты, имитирующие поведение другого объекта. В качестве примера рассмотрим два следующих класса:

```
class Foo(object):
    def spam(self,a,b):
        pass

class FooProxy(object):
    def __init__(self,f):
        self.f = f
    def spam(self,a,b):
        return self.f.spam(a,b)
```

В этом примере класс `FooProxy` функционально идентичен классу `Foo`. Он реализует те же самые методы, а его внутренняя реализация даже использует класс `Foo`. Однако для системы типов классы `FooProxy` и `Foo` являются совершенно разными. Например:

```
f = Foo()          # Создаст объект класса Foo
g = FooProxy(f)    # Создаст объект класса FooProxy
isinstance(g, Foo) # Вернет False
```

Если в программе будет предусмотрена явная проверка на принадлежность к классу `Foo` с помощью функции `isinstance()`, программа определенно не будет работать с объектом `FooProxy`. Однако в действительности совсем не всегда требуется соблюдение такой степени строгости. Иногда бывает достаточно убедиться, что некоторый объект может использоваться вместо объекта `Foo`, так как он реализует точно такой же интерфейс. Сделать это возможно, создав объект, который переопределяет поведение функций `isinstance()` и `issubclass()`, позволяя сгруппировать объекты вместе и проверить их типы. Например:

```
class IClass(object):
    def __init__(self):
        self.implementors = set()
    def register(self,C):
        self.implementors.add(C)
    def __instancecheck__(self,x):
        return self.__subclasscheck__(type(x))
    def __subclasscheck__(self,sub):
        return any(c in self.implementors for c in sub.mro())

# Пример использования объекта
IFoo = IClass()
IFoo.register(Foo)
IFoo.register(FooProxy)
```

В этом примере создается объект класса `IClass`, который просто объединяет в множество коллекцию разных классов. Метод `register()` добавляет в множество новый класс. Специальный метод `__instancecheck__()` вызывается при выполнении любой операции `isinstance(`*`x`*`, IClass)`. Специальный метод `__subclasscheck__()` вызывается при выполнении операции `issubclass(`*`C`*`,IClass)`.

Теперь с помощью объекта IFoo с зарегистрированными в нем классами можно выполнять проверку типов, как показано ниже:

```
f = Foo()                  # Создаст объект класса Foo
g = FooProxy(f)            # Создаст объект класса FooProxy
isinstance(f, IFoo)        # Вернет True
isinstance(g, IFoo)        # Вернет True
issubclass(FooProxy, IFoo) # Вернет True
```

Следует отметить, что в этом примере не производится строгая проверка типов. Операции проверки экземпляров в объекте IFoo перегружены так, что проверяют лишь принадлежность к группе классов. Здесь не делается никаких утверждений о фактическом программном интерфейсе и не выполняется никаких других проверок. В действительности, в этом объекте можно зарегистрировать любые классы по своему выбору, независимо от отношений между ними. Обычно группировка классов основана на некотором критерии, например все классы реализуют один и тот же программный интерфейс. Однако нельзя с уверенностью делать подобные выводы при использовании перегруженных методов __instancecheck__() или __subclasscheck__(). Фактическая интерпретация этого факта остается за приложением.

В языке Python имеется более формальный механизм группировки объектов, определения интерфейсов и проверки типов. Все это может быть достигнуто за счет определения абстрактных базовых классов, о которых рассказывается в следующем разделе.

Абстрактные базовые классы

В предыдущем разделе была продемонстрирована возможность перегрузки операций isinstance() и issubclass(). Благодаря ей можно создавать объекты, группирующие похожие классы и выполняющие различные операции по проверке типов. *Абстрактные базовые классы* построены на основе этой концепции и реализуют механизм организации объектов в иерархии, позволяющий утверждать о наличии требуемых методов и делать другие выводы.

Для определения абстрактного базового класса используется модуль abc. Этот модуль определяет метакласс (ABCMeta) и группу декораторов (@abstractmethod и @abstractproperty), использование которых демонстрируется ниже:

```
from abc import ABCMeta, abstractmethod, abstractproperty
class Foo:                  # В Python 3 используется синтаксис
    __metaclass__ = ABCMeta # class Foo(metaclass=ABCMeta)
    @abstractmethod
    def spam(self,a,b):
        pass
    @abstractproperty
    def name(self):
        pass
```

В определении абстрактного класса должна быть объявлена ссылка на метакласс ABCMeta, как показано выше (обратите также внимание на разли-

чия в синтаксисе между Python 2 и 3). Это совершенно необходимо, потому что реализация абстрактных классов опирается на метаклассы (подробнее о метаклассах рассказывается в следующем разделе). Внутри абстрактного класса определения методов и свойств, которые должны быть реализованы в подклассах, выполняются с помощью декораторов `@abstractmethod` и `@abstractproperty` класса `Foo`.

Абстрактный класс не может использоваться непосредственно для создания экземпляров. Если попытаться создать экземпляр предыдущего класса `Foo`, будет возбуждено исключение:

```
>>> f = Foo()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't instantiate abstract class Foo with abstract methods spam
(Перевод:
Трассировочная информация (самый последний вызов - самый нижний):
  Файл "<stdin>", строка 1, в <module>
TypeError: Невозможно создать экземпляр абстрактного класса Foo с абстрактным
методом spam
)
>>>
```

Это ограничение переносится и на производные классы. Например, если представить, что имеется класс `Bar`, наследующий класс `Foo`, но не имеющий реализации одного или более абстрактных методов, то попытка создать экземпляр класса `Bar` завершится неудачей с аналогичной ошибкой. Благодаря этой дополнительной проверке абстрактные классы являются удобным инструментом, когда необходимо гарантировать реализацию свойств и методов в подклассах.

Абстрактный класс определяет, какие свойства и методы должны быть реализованы в подклассах, но он не предъявляет никаких требований к аргументам или возвращаемым значениям. То есть абстрактный класс не может потребовать, чтобы метод в подклассе принимал те же самые аргументы, что и абстрактный метод. То же относится и к свойствам – абстрактный класс требует определить свойство, но не требует, чтобы в подклассе была реализована поддержка тех же самых операций над свойством (`get`, `set` и `delete`), что и в базовом классе.

Абстрактный класс не может использоваться для создания экземпляров, но он может определять свойства и методы для использования в подклассах. Кроме того, из подкласса допускается вызывать методы, которые были объявлены абстрактными в базовом классе. Например, вызов `Foo.spam(a,b)` в подклассе считается допустимым.

Абстрактные базовые классы позволяют регистрировать существующие классы как наследующие этот базовый класс. Делается это с помощью метода `register()`, как показано ниже:

```
class Grok(object):
    def spam(self,a,b):
        print("Grok.spam")
```

```
Foo.register(Grok)    # Зарегистрирует Grok, как наследующий
                      # абстрактный базовый класс Foo
```

Когда производится регистрация класса в некотором абстрактном базовом классе, операции проверки типа (такие как `isinstance()` и `issubclass()`) с привлечением абстрактного базового класса будут возвращать `True` для экземпляров зарегистрированных классов. В ходе регистрации класса в абстрактном базовом классе не проверяется, действительно ли регистрируемый класс реализует абстрактные свойства и методы. Процедура регистрации оказывает влияние только на операции проверки типа и не производит дополнительных проверок на наличие ошибок в регистрируемом классе.

В отличие от многих других объектно-ориентированных языков программирования, встроенные типы в языке Python организованы в виде достаточно простой иерархии. Например, если взглянуть на такие встроенные типы, как `int` или `float`, можно заметить, что они являются прямыми наследниками класса `object`, родоначальника всех объектов, а не какого-то промежуточного базового класса, представляющего числа. Это усложняет разработку программ, которые проверяют и манипулируют объектами, опираясь на их принадлежность к некоторой категории, такой как числа.

Механизм абстрактных классов позволяет решить эту проблему за счет включения существующих объектов в иерархии типов, определяемых пользователем. Более того, некоторые библиотечные модули организуют встроенные типы по категориям, в соответствии с различными особенностями, которыми они обладают. Модуль `collections` содержит абстрактные базовые классы, реализующие различные операции над последовательностями, множествами и словарями. Модуль `numbers` содержит абстрактные базовые классы, которые могут использоваться для организации иерархии числовых типов. Дополнительные подробности по этой теме приводятся в главе 14 «Математика» и в главе 15 «Структуры данных, алгоритмы и утилиты».

Метаклассы

Когда программа на языке Python объявляет класс, само определение этого класса становится объектом. Например:

```
class Foo(object): pass
isinstance(Foo,object) # Вернет True
```

Если задуматься над этим примером, можно понять, что что-то должно было создать объект `Foo`. Созданием объектов такого рода управляет специальный объект, который называется *метаклассом*. Проще говоря, метакласс – это объект, который знает, как создавать классы и управлять ими.

В предыдущем примере метаклассом, под управлением которого создается объект `Foo`, является класс с именем `type()`. Если попытаться вывести тип объекта `Foo`, можно увидеть, что он имеет тип `type`:

```
>>> type(Foo)
<type 'type'>
```

Когда с помощью инструкции `class` определяется новый класс, выполняется определенный набор действий. Во-первых, тело класса выполняется интерпретатором, как последовательность инструкций, с использованием отдельного частного словаря. Инструкции выполняются точно так же, как и в обычном программном коде, кроме того, что дополнительно производится изменение имен частных членов класса (начинающихся с префикса `__`). В заключение имя класса, список базовых классов и словарь передаются конструктору метакласса, который создает соответствующий объект класса. Следующий пример демонстрирует, как это делается:

```
class_name = "Foo"        # Имя класса
class_parents = (object,) # Базовые классы
class_body = """          # Тело класса
def __init__(self,x):
    self.x = x
def blah(self):
    print("Hello World")
"""
class_dict = { }
# Выполнить тело класса с использованием локального словаря class_dict
exec(class_body,globals(),class_dict)

# Создать объект класса Foo
Foo = type(class_name,class_parents,class_dict)
```

Заключительный этап создания класса, когда вызывается метакласс `type()`, можно изменить. Повлиять на события, происходящие на заключительном этапе определения класса, можно несколькими способами. Например, в классе можно явно указать его метакласс либо установив переменную класса `__metaclass__` (в Python 2), либо добавив именованный аргумент `metaclass` в кортеж с именами базовых классов (в Python 3).

```
class Foo:               # В Python 3 используется синтаксис
    __metaclass__ = type # class Foo(metaclass=type)
...
```

Если метакласс явно не указан, инструкция `class` проверит первый элемент в кортеже базовых классов (если таковой имеется). В этом случае метаклассом будет тип первого базового класса. То есть инструкция

```
class Foo(object): pass
```

создаст объект `Foo` того же типа, которому принадлежит класс `object`.

Если базовые классы не указаны, инструкция `class` проверит наличие глобальной переменной с именем `__metaclass__`. Если такая переменная присутствует, она будет использоваться при создании классов. С помощью этой переменной можно управлять созданием классов, когда используется простая инструкция `class`. Например:

```
__metaclass__ = type
class Foo:
    pass
```

Наконец, если переменная `__metaclass__` не будет найдена, интерпретатор будет использовать метакласс по умолчанию. В **Python 2 таким метаклассом** по умолчанию является `types.ClassType`, который известен, как класс *старого стиля*. Этот вид класса не рекомендуется к использованию, начиная с версии **Python 2.2, и соответствует оригинальной реализации классов** в языке Python. Эти классы по-прежнему поддерживаются, но они не должны использоваться в новых программах и в этой книге рассматриваться не будут. В Python 3 метаклассом по умолчанию является `type`.

В основном метаклассы используются во фреймворках, когда требуется более полный контроль над определениями пользовательских объектов. Когда определяется нестандартный метакласс, он обычно наследует класс `type` и переопределяет такие методы, как `__init__()` или `__new__()`. Ниже приводится пример метакласса, который требует, чтобы все методы снабжались строками документирования:

```
class DocMeta(type):
    def __init__(self,name,bases,dict):
        for key, value in dict.items():
            # Пропустить специальные и частные методы
            if key.startswith("__"): continue
            # Пропустить любые невызываемые объекты
            if not hasattr(value,"__call__"): continue
            # Проверить наличие строки документирования
            if not getattr(value,"__doc__"):
                raise TypeError("%s must have a docstring" % key)
        type.__init__(self,name,bases,dict)
```

В этом метаклассе реализован метод `__init__()`, который проверяет содержимое словаря класса. Он отыскивает в словаре методы и проверяет, имеют ли они строки документирования. Если в каком-либо методе строка документирования отсутствует, возбуждается исключение `TypeError`. В противном случае для инициализации класса вызывается реализация метода `type.__init__()`.

Чтобы воспользоваться этим метаклассом, класс должен явно выбрать его. Обычно для этой цели сначала определяется базовый класс, такой, как показано ниже:

```
class Documented:           # В Python 3 используется синтаксис
    __metaclass__ = DocMeta # class Documented(metaclass=DocMeta)
```

А затем этот базовый класс используется, как родоначальник всех объектов, которые должны включать в себя описание. Например:

```
class Foo(Documented):
    spam(self,a,b):
        "Метод spam делает кое-что"
        pass
```

Этот пример иллюстрирует одно из основных применений метаклассов, состоящий в проверке и сборе информации об определениях классов. Метакласс ничего не изменяет в создаваемом классе, он просто выполняет некоторые дополнительные проверки.

В более сложных случаях перед тем, как создать класс, метакласс может не только проверять, но и изменять содержимое его определения. Если предполагается вносить какие-либо изменения, необходимо переопределить метод `__new__()`, который выполняется перед созданием класса. Этот прием часто объединяется с приемом обертывания атрибутов дескрипторами или свойствами, потому что это единственный способ получить имена, использованные в классе. В качестве примера ниже приводится модифицированная версия дескриптора `TypedProperty`, который был реализован в разделе «Дескрипторы»:

```
class TypedProperty(object):
    def __init__(self,type,default=None):
        self.name = None
        self.type = type
        if default: self.default = default
        else:       self.default = type()
    def __get__(self,instance,cls):
        return getattr(instance,self.name,self.default)
    def __set__(self,instance,value):
        if not isinstance(value,self.type):
            raise TypeError("Значение должно быть типа %s" % self.type)
        setattr(instance,self.name,value)
    def __delete__(self,instance):
        raise AttributeError("Невозможно удалить атрибут")
```

В данном примере атрибуту `name` дескриптора просто присваивается значение `None`. Заполнение этого атрибута будет поручено метаклассу. Например:

```
class TypedMeta(type):
    def __new__(cls,name,bases,dict):
        slots = [ ]
        for key,value in dict.items():
            if isinstance(value,TypedProperty):
                value.name = "_" + key
                slots.append(value.name)
        dict['__slots__'] = slots
        return type.__new__(cls,name,bases,dict)

# Базовый класс для объектов, определяемых пользователем
class Typed:                    # В Python 3 используется синтаксис
    __metaclass__ = TypedMeta # class Typed(metaclass=TypedMeta)
```

В этом примере метакласс просматривает словарь класса с целью отыскать экземпляры класса `TypedProperty`. В случае обнаружения такого экземпляра он устанавливает значение атрибута `name` и добавляет его в список имен `slots`. После этого в словарь класса добавляется атрибут `__slots__` и вызывается метод `__new__()` метакласса `type`, который создает объект класса. Ниже приводится пример использования нового метакласса:

```
class Foo(Typed):
    name = TypedProperty(str)
    num = TypedProperty(int,42)
```

Метаклассы способны коренным образом изменять поведение и семантику пользовательских классов. Однако не следует злоупотреблять этой

возможностью и изменять поведение классов так, чтобы оно существенно отличалось от поведения, описанного в стандартной документации. Пользователи будут обескуражены, если создаваемые ими классы не будут придерживаться обычных правил программирования для классов.

Декораторы классов

В предыдущем примере было показано, как с помощью метаклассов можно управлять процессом создания классов. Однако иногда бывает достаточно выполнить некоторые действия уже после того, как класс будет определен, например добавить класс в реестр или в базу данных. Альтернативный подход к решению подобных задач заключается в использовании декоратора класса. *Декоратор класса* – это функция, которая принимает и возвращает класс. Например:

```
registry = { }
def register(cls):
    registry[cls.__clsid__] = cls
    return cls
```

В этом примере функция `register` отыскивает в классе атрибут `__clsid__`. Если этот атрибут определен, он используется для добавления класса в словарь, который служит для отображения идентификаторов классов в объекты классов. Эту функцию можно использовать как декоратор, поместив его непосредственно перед определением класса. Например:

```
@register
class Foo(object):
    __clsid__ = "123-456"
    def bar(self):
        pass
```

Здесь к синтаксису с использованием декоратора мы прибегли, главным образом, ради удобства. Того же самого результата можно добиться другим способом:

```
class Foo(object):
    __clsid__ = "123-456"
    def bar(self):
        pass
register(Foo) # Зарегистрировать класс
```

Можно до бесконечности придумывать всякие дьявольские ухищрения, которые можно было бы реализовать в функции-декораторе класса, однако лучше все-таки избегать чрезмерных превращений, таких как создание обертки вокруг класса или переопределение его содержимого.

8

Модули, пакеты и дистрибутивы

Крупные программы на языке Python обычно организованы в виде набора модулей и пакетов. Кроме того, большое число модулей также входит в стандартную библиотеку Python. В этой главе более подробно рассказывается о системе модулей и пакетов. Помимо этого здесь приводится информация о том, как устанавливать модули сторонних разработчиков и как создавать дистрибутивы с исходными текстами.

Модули и инструкция import

Любой файл с исходными текстами на языке Python может использоваться как модуль. В качестве примера рассмотрим содержимое файла `spam.py`:

```
# spam.py
a = 37
def foo():
    print("Функция foo(), a = %s" % a)
def bar():
    print("Функция bar(), вызывается функция foo()")
    foo()
class Spam(object):
    def grok(self):
        print("Метод Spam.grok")
```

Загрузить файл с этим исходным программным кодом, как модуль, можно с помощью инструкции `import spam`. Когда инструкция `import` впервые загружает модуль, она выполняет следующие три операции:

1. Создает новое пространство имен, которое будет служить контейнером для всех объектов, определяемых в соответствующем файле. Это – то пространство имен, которое будет доступно функциям и методам, объявленным в модуле, использующим инструкцию `global`.
2. Выполняет программный код в модуле внутри вновь созданного пространства имен.

3. Создает в вызывающей программе имя, ссылающееся на пространство имен модуля. Это имя совпадает с именем модуля и используется, как показано ниже:

```
import spam     # Загрузит и выполнит модуль 'spam'
x = spam.a      # Обратится к переменной модуля 'spam'
spam.foo()      # Вызовет функцию в модуле 'spam'
s = spam.Spam() # Создаст экземпляр класса spam.Spam()
s.grok()
...
```

Важно подчеркнуть, что инструкция `import` выполнит все инструкции в загруженном файле. Если в дополнение к объявлению переменных, функций и классов в модуле содержатся некоторые вычисления и вывод результатов, то результаты будут выведены на экран в момент загрузки модуля. Кроме этого, обычное недопонимание работы с модулями связано со способом обращения к классам, объявленным в нем. Запомните, что если в файле `spam.py` объявлен класс `Spam`, обращаться к этому классу следует по имени `spam.Spam`.

Чтобы импортировать сразу несколько модулей, их имена можно перечислить в инструкции `import` через запятую, например:

```
import socket, os, re
```

Имя, используемое для ссылки на модуль, можно изменить с помощью квалификатора `as`. Например:

```
import spam as sp
import socket as net
sp.foo()
sp.bar()
net.gethostname()
```

Когда модуль загружается под другим именем, как в примере выше, новое имя можно использовать только в файле или в контексте, где находится инструкция `import`. Другие модули по-прежнему могут загружать модуль под его оригинальным именем.

Возможность изменения имени импортируемого модуля может оказаться полезным инструментом, позволяющим писать расширяемый программный код. Например, допустим, что имеется два модуля, `xmlreader.py` и `csvreader.py`, каждый из которых объявляет функцию `read_data(`*`filename`*`)`, которая читает некоторые данные из файла, но работает с файлами разных форматов. Можно написать такой программный код, который будет выбирать необходимый модуль чтения, как показано ниже:

```
if format == 'xml':
    import xmlreader as reader
elif format == 'csv':
    import csvreader as reader
data = reader.read_data(filename)
```

Модули в языке Python – это «первоклассные» объекты. То есть они могут присваиваться переменным, помещаться в структуры данных, такие как списки, и передаваться между частями программы в виде элемента

данных. Например, переменная reader в предыдущем примере – это просто ссылка на соответствующий объект модуля. С точки зрения внутренней реализации объект модуля – это словарь, который хранит содержимое пространства имен модуля. Этот словарь доступен как атрибут __dict__ модуля, и всякий раз, когда производится обращение к значению внутри модуля, вы фактически работаете с этим словарем.

Инструкция import может появляться в любом месте программы. Однако программный код любого модуля загружается и выполняется только один раз, независимо от количества инструкций import, загружающих его. Все последующие инструкции import будут просто связывать имя модуля с объектом модуля, созданным первой встретившейся инструкцией import. Словарь, содержащий все модули, загруженные к текущему моменту, доступен в виде переменной sys.modules. Этот словарь отображает имена модулей на объекты модулей. Содержимое этого словаря используется, чтобы определить, должна ли инструкция import загружать свежую копию модуля.

Импортирование отдельных имен из модулей

Для загрузки отдельных определений из модуля в текущее пространство имен используется инструкция from. По своей функциональности инструкция from идентична инструкции import, за исключением того, что вместо создания имени, ссылающегося на вновь созданное пространство имен модуля, она помещает ссылки на один или более объектов, объявленных в модуле, в текущее пространство имен:

```
from spam import foo # Импортирует модуль spam и помещает имя 'foo'
                     # в текущее пространство имен
foo()                # Вызовет spam.foo()
spam.foo()           # NameError: spam
```

Инструкция from также может принимать список имен объектов, разделенных запятыми. Например:

```
from spam import foo, bar
```

Если список импортируемых имен достаточно длинный, его можно заключить в круглые скобки. Благодаря этому инструкцию import можно разместить в нескольких строках программы. Например:

```
from spam import (foo,
                  bar,
                  Spam)
```

Кроме того, в инструкции from можно использовать квалификатор as, чтобы присвоить импортируемым объектам другие имена. Например:

```
from spam import Spam as Sp
s = Sp()
```

Помимо этого, для загрузки всех определений, имеющихся в модуле, за исключением тех, имена которых начинаются с символа подчеркивания, можно использовать шаблонный символ звездочки (*). Например:

```
from spam import * # Загрузит все определения в текущее пространство имен
```

Инструкция `from module import *` может использоваться только для загрузки модулей верхнего уровня. В частности, инструкцию импортирования в таком виде нельзя использовать внутри функций – из-за ограничений, накладываемых правилами видимости (например, когда функция компилируется в байт-код, все имена, используемые в функции, должны быть квалифицированы полностью).

Модули позволяют организовать более точное управление набором имен, импортируемых инструкцией `from module import *`, посредством объявления списка `__all__`. Например:

```
# module: spam.py
__all__ = [ 'bar', 'Spam' ] # Эти имена будут импортироваться инструкцией
                            # from spam import *
```

Правила видимости для определений, импортированных инструкцией `from`, не изменяются. Например, рассмотрим следующий фрагмент:

```
from spam import foo
a = 42
foo()     # Выведет "Функция foo(), a = 37"
```

В данном примере функция `foo()`, объявленная в модуле `spam.py`, обращается к глобальной переменной `a`. Когда ссылка на функцию `foo()` помещается в другое пространство имен, это не приводит к изменению правил видимости переменных внутри функции. То есть глобальным пространством имен для функции всегда будет модуль, в котором она была объявлена, а не пространство имен, в которое эта функция была импортирована и откуда была вызвана. То же самое относится и к вызовам функций. Например, в следующем фрагменте вызывается функция `bar()`, которая в свою очередь вызывает функцию `spam.foo()`, а не переопределенную функцию `foo()`, объявленную в следующем примере:

```
from spam import bar
def foo():
    print("Другая функция foo")
bar()     # Когда функция bar() вызовет функцию foo(), то будет вызвана
          # spam.foo(), а не функция foo(), объявленная выше
```

Другой типичный источник ошибок, связанных с инструкцией `from`, касается поведения глобальных переменных. Рассмотрим в качестве примера следующий фрагмент:

```
from spam import a, foo # Импортирует глобальную переменную
a = 42                  # Изменит значение переменной
foo()                   # Выведет "Функция foo(), a = 37"
print(a)                # Выведет "42"
```

Здесь важно понять, что инструкция присваивания значения переменной в языке **Python не является операцией сохранения значения. То есть** инструкция присваивания в предыдущем примере не сохраняет новое значение в переменной `a`, затирая старое значение. Вместо этого создается новый объект, содержащий значение `42`, а в переменную `a` записывается ссылка

на него. После этого переменная a потеряет связь со значением, импортированным из модуля, и будет ассоциирована с некоторым другим объектом. По этой причине невозможно с помощью инструкции from имитировать поведение глобальных переменных или общих блоков, такое, как в языках программирования **C или Fortran. Если в программе потребуется иметь изменяемые глобальные параметры, их можно поместить в модуль и явно использовать имя модуля, импортированного с помощью инструкции** import **(то есть явно использовать имя** spam.a**).**

Выполнение модуля как самостоятельной программы

Интерпретатор Python может выполнять файлы с программным кодом двумя способами. Инструкция import выполняет программный код библиотечного модуля в собственном пространстве имен. Однако программный код может также выполняться как отдельная программа или сценарий. Это происходит, когда имя файла передается интерпретатору как имя сценария:

```
% python spam.py
```

Для каждого модуля определяется переменная __name__, где хранится имя модуля. Программы могут проверять значение этой переменной, чтобы определить, в каком режиме они выполняются. При выполнении интерпретатором модуль верхнего уровня получает имя __main__. Программа, имя которой было передано интерпретатору в виде аргумента командной строки, или введенная в интерактивной оболочке, выполняется внутри пространства имен модуля __main__. Иногда бывает необходимо изменить поведение программы, в зависимости от того, была ли она импортирована как модуль или запущена в пространстве имен __main__. Например, модуль может содержать какие-либо тесты, которые должны выполняться при запуске модуля как отдельной программы и не должны выполняться, когда модуль импортируется другим модулем. Реализовать это можно, как показано ниже:

```
# Проверить, был ли модуль запущен как программа
if __name__ == '__main__':
    # Да
    инструкции
else:
    # Нет, файл был импортирован как модуль
    инструкции
```

В файлы с программным кодом, которые могут использоваться как библиотеки, часто включается дополнительный программный код, с примерами или тестами. Например, при разработке модуля можно поместить программный код, производящий тестирование библиотеки, внутрь инструкции if, как показано выше, и просто запускать модуль, как самостоятельную программу, чтобы выполнить его. Этот программный код не будет выполняться, когда пользователь будет импортировать библиотеку как модуль.

Путь поиска модулей

Чтобы загрузить модуль, интерпретатор просматривает каталоги, список которых находится в переменной `sys.path`. Первым элементом в списке `sys.path` обычно является пустая строка `''`, которая обозначает текущий рабочий каталог. Остальные элементы списка `sys.path` могут содержать имена каталогов, архивных файлов в формате `.zip` и файлов пакетов с расширением `.egg`. Порядок следования элементов в `sys.path` определяет порядок поиска файла при загрузке модуля. Чтобы добавить элемент в путь поиска, достаточно просто добавить его в этот список.

Несмотря на то что путь обычно включает имена каталогов, в путь поиска можно также добавлять **zip-архивы, содержащие модули Python. Эта особенность** может использоваться для упаковки коллекции модулей в единый файл. Например, предположим, что вы создали два модуля, `foo.py` и `bar.py`, и поместили их в zip-файл с именем `mymodules.zip`. Этот файл можно будет добавить в путь поиска Python, как показано ниже:

```
import sys
sys.path.append("mymodules.zip")
import foo, bar
```

Также допускается указывать конкретное местоположение в структуре каталогов, внутри **zip-файла. Кроме того, zip-файлы можно указывать вперемежку** с обычными компонентами пути в файловой системе. Например:

```
sys.path.append("/tmp/modules.zip/lib/python")
```

Кроме `.zip`-файлов, в путь поиска можно также добавлять файлы с расширением `.egg`. Файлы с расширением `.egg` – это пакеты, созданные с помощью библиотеки `setuptools`. С этим распространенным форматом пакетов вы встретитесь, когда будете устанавливать сторонние библиотеки и расширения Python. Файлы с расширением `.egg` – это, по сути, обычные zip-файлы, содержащие дополнительные метаданные (например, номер версии, список зависимостей и так далее). То есть данные, находящиеся в файле с расширением `.egg`, можно извлекать с помощью стандартных инструментов, предназначенных для работы с zip-файлами.

Несмотря на наличие возможности импортировать zip-файлы, они имеют некоторые ограничения, о которых следует знать. Во-первых, из архивов можно импортировать только файлы с расширениями `.py`, `.pyw`, `.pyc` и `.pyo`. Разделяемые библиотеки и модули расширений, написанные на языке C, не могут загружаться из архивов, хотя некоторые системы пакетов, такие как `setuptools`, предоставляют обходные пути решения этой проблемы (обычно извлекая расширения на языке **C во временный каталог и загружая** модули из него). Кроме того, интерпретатор Python не будет создавать файлы с расширениями `.pyc` и `.pyo`, если файлы `.py` были загружены из архива (пояснения в следующем разделе). Поэтому очень важно убедиться, что эти файлы были созданы заранее и упакованы в архив, чтобы не иметь потерь в производительности при загрузке модулей.

Загрузка и компиляция модулей

До сих пор в этой главе рассматривались модули, которые представляют собой файлы с программным кодом исключительно на языке Python. Однако в действительности модули, загружаемые инструкцией `import`, могут быть отнесены к четырем основным категориям:

- Программный код на языке Python (файлы с расширением `.py`)
- Расширения на языке C или C++, которые были скомпилированы в виде разделяемых библиотек или DLL
- Пакеты, содержащие коллекции модулей
- Встроенные модули, написанные на языке C и скомпонованные с интерпретатором Python

При поиске модуля (например, `foo`) интерпретатор просматривает каждый каталог, упомянутый в списке `sys.path`, в поисках следующих файлов (перечисленных в том порядке, в каком ведется поиск):

1. Каталог `foo`, объявленный как пакет.
2. `foo.pyd`, `foo.so`, `foomodule.so` или `foomodule.dll` (скомпилированные расширения).
3. `foo.pyo` (только если при компиляции был использован ключ `-O` или `-OO`).
4. `foo.pyc`.
5. `foo.py` (в Windows интерпретатор также проверяет наличие файлов с расширением `.pyw`).

Пакеты будут описаны чуть ниже; скомпилированные расширения описаны в главе 26 «Расширение и встраивание Python». Что касается файлов с расширением `.py`, то когда модуль импортируется впервые, он компилируется в байт-код и сохраняется на диске в файле с расширением `.pyc`. При всех последующих обращениях к импортированию этого модуля интерпретатор будет загружать скомпилированный байт-код, если только с момента создания байт-кода в файл `.py` не вносились изменения (в этом случае файл `.pyc` будет создан заново). Файлы `.pyo` создаются, когда интерпретатор запускается с ключом `-O`. В этом случае из байт-кода удаляются номера строк, инструкции `assert` и прочая отладочная информация. В результате файлы получаются меньше и выполняются интерпретатором немного быстрее. Если вместо ключа `-O` использовать ключ `-OO`, из файла также будут удалены строки документирования. Удаление строк документирования происходит именно при создании файлов с расширением `.pyo`, но не во время их загрузки. Если интерпретатору не удастся обнаружить ни один из этих файлов в каталогах, перечисленных в списке `sys.path`, он проверит, не соответствует ли это имя встроенному модулю. Если такого встроенного модуля не существует, будет возбуждено исключение `ImportError`.

Автоматическая компиляция программного кода в файлы с расширениями `.pyc` и `.pyo` производится только при использовании инструкции `import`. При запуске программ из командной строки эти файлы не создаются. Кро-

ме того, эти файлы не создаются, если каталог, содержащий файл .py модуля, не доступен для записи (например, из-за нехватки прав у пользователя или когда модуль находится в zip-архиве). Запретить создание этих файлов можно также, запустив интерпретатор с ключом -B.

Если имеются файлы .pyc и .pyo, файлы .py с исходными текстами могут отсутствовать. То есть, если вы не желаете включать в пакет исходные тексты модулей, можно просто упаковать набор файлов .pyc. Однако следует помнить, что интерпретатор Python обладает мощной возможностью интроспекции и дизассемблирования. Опытные пользователи без особого труда смогут исследовать ваши модули и узнать массу подробностей о них, даже в случае отсутствия исходных текстов. Кроме того, не нужно забывать, что файлы .pyc могут зависеть от версии интерпретатора. То есть файл .pyc, созданный одной версией интерпретатора Python, может оказаться нерабочим с будущими версиями.

Когда инструкция import пытается отыскать файлы, поиск производится с учетом регистра символов в именах файлов, даже в операционных системах, где файловые системы не различают регистр символов, в таких как Windows и OS X (хотя такие системы сохраняют регистр символов). Поэтому инструкция import foo сможет импортировать только файл foo.py и не сможет —FOO.PY. Вообще желательно избегать использования символов различных регистров в именах модулей.

Выгрузка и повторная загрузка модулей

В языке Python отсутствует полноценная поддержка возможности повторной загрузки или выгрузки модулей, импортированных ранее. Даже если удалить модуль из словаря sys.modules, в общем случае это не приведет к выгрузке модуля из памяти. Это может быть обусловлено наличием ссылок на объект модуля в других компонентах программы, которые загрузили этот модуль с помощью инструкции import. Более того, если в программе были созданы экземпляры классов, объявленных в модуле, эти экземпляры содержат ссылки на соответствующие им объекты классов, которые в свою очередь хранят ссылки на модуль, в котором они были объявлены.

Наличие ссылок на модуль в разных местах программы делает практически невозможной повторную загрузку модуля после внесения изменений в его реализацию. Например, если удалить модуль из словаря sys.modules и вызвать инструкцию import, чтобы опять загрузить его, это не окажет влияния на ссылки, которые были созданы в программе до этого. Вместо этого вы получите ссылку на новый модуль, созданный самой последней инструкцией import, и множество ссылок на старую версию модуля из других мест программы. Скорее всего, окажется нежелательным и наверняка небезопасным использование в реальных условиях реально функционирующего кода, в котором нет полного контроля над средой выполнения.

Старые версии Python предоставляли функцию reload(), позволяющую выгружать модули. Однако эта функция никогда не была безопасной в использовании (по всем вышеупомянутым причинам), и ее настоятельно ре-

комендовали использовать исключительно для отладки. В версии Python 3 такая возможность была полностью ликвидирована. Поэтому **лучше никогда не полагаться на нее.**

Наконец, следует отметить, что расширения для Python, написанные на языке C/C++, не могут быть безопасно выгружены и повторно загружены никаким способом. Такая возможность никогда не поддерживалась и не поддерживается, к тому же этому может препятствовать сама операционная система. Поэтому ваша единственная возможность – это полностью перезапустить процесс интерпретатора Python.

Пакеты

Пакеты позволяют сгруппировать коллекцию модулей под общим именем пакета. Этот прием позволяет решить проблему конфликтов имен между именами модулей, используемых в различных приложениях. Пакет создается как каталог с тем же именем, в котором создается файл с именем `__init__.py`. Затем в этом каталоге можно сохранить исходные файлы, скомпилированные расширения и подпакеты. Например, пакет может иметь такую структуру:

```
Graphics/
    __init__.py
    Primitive/
        __init__.py
        lines.py
        fill.py
        text.py
        ...
    Graph2d/
        __init__.py
        plot2d.py
        ...
    Graph3d/
        __init__.py
        plot3d.py
        ...
    Formats/
        __init__.py
        gif.py
        png.py
        tiff.py
        jpeg.py
```

Загрузка модулей из пакета может быть выполнена с помощью инструкции `import` несколькими способами:

- `import Graphics.Primitive.fill`

  Загрузит модуль `Graphics.Primitive.fill`. Объекты, содержащиеся в этом модуле, получат имена, такие как `Graphics.Primitive.fill.floodfill(img,x,y,color)`.

- `from Graphics.Primitive import fill`

  Загрузит модуль `fill`, но сделает возможным обращаться к нему без использования префикса с именем пакета; например, `fill.floodfill(img,x,y,color)`.

- `from Graphics.Primitive.fill import floodfill`

  Загрузит модуль `fill`, но сделает возможным обращаться к функции `floodfill` непосредственно; например, `floodfill(img,x,y,color)`.

Всякий раз когда какая-либо часть пакета импортируется впервые, выполняется программный код в файле `__init__.py`. Этот файл может быть пустым, но может также содержать программный код, выполняющий инициализацию пакета. Выполнены будут все файлы `__init__.py`, которые встретятся инструкции `import` в процессе ее выполнения. То есть инструкция `import Graphics.Primitive.fill`, показанная выше, сначала выполнит файл `__init__.py` в каталоге `Graphics`, а затем файл `__init__.py` в каталоге `Primitive`.

Существует одна проблема, характерная для пакетов, связанная с выполнением такой инструкции:

```
from Graphics.Primitive import *
```

Когда программист использует подобную инструкцию, он обычно хочет импортировать все модули, ассоциированные с пакетом, в текущее пространство имен. Однако из-за того, что соглашения по именованию файлов могут изменяться от системы к системе (особенно это относится к регистру символов), **Python не в состоянии точно определить, какие модули должны загружаться.** Как результат, эта инструкция просто импортирует все имена, которые определены в файле `__init__.py`, находящемся в каталоге `Primitive`. Это поведение можно изменить, определив список `__all__`, содержащий имена всех модулей в пакете. Этот список должен быть объявлен в файле `__init__.py` пакета, например:

```
# Graphics/Primitive/__init__.py
__all__ = ["lines","text","fill"]
```

Теперь, когда интерпретатор встретит инструкцию `from Graphics.Primitive import *`, он загрузит все перечисленные модули, как и следовало ожидать.

Еще одна малозаметная проблема, связанная с пакетами, касается модулей, которые должны импортировать другие модули из того же пакета. Например, допустим, что модуль `Graphics.Primitive.fill` должен импортировать модуль `Graphics.Primitive.lines`. Для этого можно просто указать полностью квалифицированное имя модуля (например, `from Graphics.Primitives import lines`) или выполнить импортирование относительно текущего пакета, например:

```
# fill.py
from . import lines
```

В этом примере символ `.` в инструкции `from . import lines` ссылается на тот же каталог, где находится вызывающий модуль. То есть данная инструк-

ция будет искать модуль lines в том же каталоге, где находится файл fill.py. При импортировании модулей из пакета следует быть особенно внимательными и не использовать инструкцию вида import *module*. В старых версиях Python было не очевидно, ссылается ли инструкция import *module* на модуль в стандартной библиотеке или на модуль в пакете. В старых версиях Python такая инструкция сначала пыталась загрузить модуль из того же каталога пакета, где находится импортирующий модуль, и только потом, в случае неудачи, делалась попытка загрузить модуль из стандартной библиотеки. Однако в Python 3, инструкция import предполагает, что указан абсолютный путь, и будет пытаться загрузить модуль из стандартной библиотеки. Использование инструкции импортирования по относительному пути более четко говорит о ваших намерениях.

Возможность импортирования по относительному пути можно также использовать для загрузки модулей, находящихся в других каталогах того же пакета. Например, если в модуле Graphics.Graph2D.plot2d потребуется импортировать модуль Graphics.Primitives.lines, инструкция импорта будет иметь следующий вид:

```
# plot2d.py
from ..Primitives import lines
```

В этом примере символы .. перемещают точку начала поиска на уровень выше в дереве каталогов, а имя Primitives перемещает ее вниз, в другой каталог пакета.

Импорт по относительному пути может выполняться только при использовании инструкции импортирования вида from *module* import *symbol*. То есть такие инструкции, как import ..Primitives.lines или import .lines, будут рассматриваться как синтаксическая ошибка. Кроме того, имя *symbol* должно быть допустимым идентификатором. Поэтому такая инструкция, как from .. import Primitives.lines, также считается ошибочной. Наконец, импортирование по относительному пути может выполняться только для модулей в пакете; не допускается использовать эту возможность для ссылки на модули, которые просто находятся в другом каталоге файловой системы.

Импортирование по одному только имени пакета не приводит к импортированию всех модулей, содержащихся в этом пакете. Например, следующий фрагмент не будет работать:

```
import Graphics
Graphics.Primitive.fill.floodfill(img,x,y,color) # Ошибка!
```

Однако, так как инструкция import Graphics выполнит файл __init__.py в каталоге Graphics, в него можно добавить инструкции импортирования по относительному пути, которые автоматически загрузят все модули, как показано ниже:

```
# Graphics/__init__.py
from . import Primitive, Graph2d, Graph3d

# Graphics/Primitive/__init__.py
from . import lines, fill, text, ...
```

Теперь инструкция `import Graphics` будет импортировать все модули и обеспечит их доступность по полностью квалифицированным именам. Еще раз подчеркну, что инструкция импортирования пакета по относительному пути должна использоваться именно так, как показано выше. Если использовать простую инструкцию, такую как `import` *`module`*, она попытается загрузить модуль из стандартной библиотеки.

Наконец, когда интерпретатор импортирует пакет, он объявляет специальную переменную `__path__`, содержащую список каталогов, в которых выполняется поиск модулей пакета (`__path__` представляет собой аналог списка `sys.path` для пакета). Переменная `__path__` доступна для программного кода в файлах `__init__.py` и изначально содержит единственный элемент с именем каталога пакета. При необходимости пакет может добавлять в список `__path__` дополнительные каталоги, чтобы изменить путь поиска модулей. Это может потребоваться в случае сложной организации дерева каталогов пакета в файловой системе, которая не совпадает с иерархией пакета.

Распространение программ и библиотек на языке Python

Чтобы подготовить дистрибутив программы для передачи ее другому лицу, необходимо воспользоваться модулем `distutils`. Для начала необходимо подготовить структуру каталогов, создать в нем файл `README`, скопировать туда документацию и файлы с исходными текстами. Обычно этот каталог будет содержать смесь библиотечных модулей, пакетов и сценариев. Модули и пакеты относятся к исходным файлам, которые загружаются инструкциями `import`. Сценарии – это файлы с программным кодом, которые будут запускаться интерпретатором, как самостоятельные программы (например, как команда `python` *`имя_сценария`*). Ниже приводится пример каталога, содержащего программный код на языке Python:

```
spam/
    README.txt
    Documentation.txt
    libspam.py        # Одиночный библиотечный модуль
    spampkg/          # Пакет вспомогательных модулей
        __init__.py
        foo.py
        bar.py
    runspam.py        # Сценарий, который запускается как: python runspam.py
```

Программный код должен быть организован так, чтобы он нормально работал при запуске интерпретатора Python в каталоге верхнего уровня. Например, если запустить интерпретатор в каталоге `spam`, он должен быть в состоянии импортировать модули, компоненты пакета и запускать сценарии без изменения каких-либо настроек Python, таких как путь поиска модулей.

После того как каталог будет подготовлен, нужно создать файл setup.py в самом верхнем каталоге (в предыдущем примере – это каталог spam) и поместить в него следующий программный код:

```
# setup.py
from distutils.core import setup

setup(name = "spam",
      version = "1.0",
      py_modules = ['libspam'],
      packages = ['spampkg'],
      scripts = ['runspam.py'],
     )
```

Аргумент py_modules в функции setup() – это список всех файлов одиночных модулей, packages – список всех каталогов пакетов и scripts – список файлов сценариев. Любой из этих аргументов может быть опущен, если ваш программный продукт не включает соответствующие ему компоненты (например, не имеет сценариев). Аргумент name – это имя пакета, а version – номер версии в виде строки.

Функция setup() может принимать ряд других аргументов, содержащих дополнительные метаданные о пакете. В табл. 8.1 перечислены наиболее часто используемые аргументы. Все аргументы принимают строковые значения, за исключением classifiers, который является списком строк, например: ['Development Status :: 4 - Beta', 'Programming Language :: Python'] (полный список аргументов можно найти по адресу *http://pypi.python.org*).

Таблица 8.1. Аргументы функции setup()

Аргумент	Описание
name	Имя пакета (обязательный)
version	Номер версии (обязательный)
author	Имя автора
author_email	Адрес электронной почты автора
maintainer	Имя лица, осуществляющего сопровождение
maintainer_email	Адрес электронной почты лица, осуществляющего сопровождение
url	Адрес домашней страницы проекта пакета
description	Краткое описание пакета
long_description	Полное описание пакета
download_url	Адрес, откуда можно загрузить пакет
classifiers	Список строк классификаторов

Файла setup.py вполне достаточно для создания дистрибутива с исходными текстами программного обеспечения. Чтобы создать дистрибутив, введите следующую команду в командной оболочке:

```
% python setup.py sdist
...
%
```

В результате в каталоге spam/dist будет создан архивный файл, такой как spam-1.0.tar.gz или spam-1.0.zip. Это и есть файл дистрибутива, который можно передать другому лицу для установки вашего программного обеспечения. Чтобы установить программу из дистрибутива, пользователь должен просто распаковать архив и выполнить следующие действия:

```
% unzip spam-1.0.zip
...
% cd spam-1.0
% python setup.py install
...
%
```

В результате программное обеспечение будет установлено на локальном компьютере и готово к использованию. Модули и пакеты обычно устанавливаются в каталог с именем "site-packages" в библиотеке Python. Чтобы определить точное местоположение этого каталога, можно проверить значение переменной sys.path. В UNIX-подобных операционных системах сценарии обычно устанавливаются в каталог установки интерпретатора Python, а в Windows – в каталог "Scripts" (обычно это каталог "C:\Python26\Scripts").

В операционной системе UNIX, если первая строка сценария начинается с последовательности символов #! и содержит текст "python", мастер установки запишет в эту строку действительный путь к интерпретатору Python на локальном компьютере. То есть, если во время разработки в сценарии был жестко задан путь к интерпретатору Python, такой как /usr/local/bin/python, после установки на другом компьютере, где Python установлен в другой каталог, этот сценарий сохранит свою работоспособность.

Файл setup.py реализует множество других команд, касающихся создания дистрибутивов. Если выполнить команду 'python setup.py bdist', будет создан двоичный дистрибутив, где все файлы .py будут скомпилированы в файлы с расширением .pyc и помещены в дерево каталогов, имитирующее структуру каталогов на локальном компьютере. Такой способ создания дистрибутивов может потребоваться, только если в приложении имеются компоненты, зависящие от типа платформы (например, если имеется расширение на языке C, которое должно распространяться в скомпилированном виде). Если в Windows запустить команду 'python setup.py bdist_wininst', будет создан исполняемый файл с расширением .exe. Если затем этот файл запустить в Windows, откроется диалог мастера установки, где пользователю будет предложено указать каталог, куда следует выполнить установку. Кроме всего прочего, при установке дистрибутивов этого типа в реестр Windows добавляются дополнительные записи, впоследствии упрощающие возможность удаления пакета.

Модуль distutils предполагает, что на компьютере пользователя уже установлен интерпретатор Python (загруженный отдельно). Безусловно, существует возможность создать выполняемый файл, который вместе с вашим

программным продуктом будет включать в себя все необходимые компоненты Python, однако рассмотрение этой темы выходит далеко за рамки данной книги (дополнительную информацию можно найти в описаниях к сторонним модулям, таким как `py2exe` или `py2app`). Если под распространение подпадают только библиотеки или простые сценарии, вам скорее всего не потребуется упаковывать в дистрибутив с программным кодом интерпретатор Python и дополнительные компоненты.

В заключение следует отметить, что модуль `distutils` обладает намного более широкими возможностями, чем было описано здесь. В главе 26 описывается, как можно использовать модуль `distutils` для сборки расширений, написанных на языке C и C++.

Несмотря на отсутствие необходимых инструментов в составе стандартной установки Python, имеется возможность распространять свое программное обеспечение в виде файлов с расширением `.egg`. Файлы этого формата создаются с помощью популярного расширения `setuptools` (*http://pypi.python.org/pypi/setuptools*). Чтобы обеспечить поддержку `setuptools`, достаточно просто изменить первую часть файла `setup.py`, как показано ниже:

```
# setup.py
try:
    from setuptools import setup
except ImportError:
    from distutils.core import setup

setup(name = "spam",
      ...
)
```

Установка сторонних библиотек

Наиболее полным ресурсом, где можно найти сторонние библиотеки и расширения для Python, является *каталог пакетов Python* (Python Package Index – PyPI), доступный по адресу *http://pypi.python.org*. Установка сторонних модулей обычно выполняется достаточно просто, но может превратиться в сложную задачу при установке крупных пакетов, зависящих от других сторонних модулей. Большинство основных расширений снабжаются программами установки для выбранной платформы, которые помогут выполнить установку, проведя вас через последовательность диалогов. Чтобы установить другие модули, обычно бывает достаточно распаковать архив, отыскать файл `setup.py` и ввести команду `python setup.py install`.

По умолчанию сторонние модули устанавливаются в каталог `site-packages`, в стандартной библиотеке Python. Для записи в этот каталог обычно требуются привилегии суперпользователя или администратора. Если вы не обладаете этими привилегиями, можно ввести команду `python setup.py install --user`, чтобы установить модуль в библиотечный каталог пользователя. В этом случае пакет будет установлен в домашний каталог пользователя, например: `"/Users/beazley/.local/lib/python2.6/site-packages"` в UNIX.

Если потребуется установить программное обеспечение в совершенно иное место, можно воспользоваться параметром `--prefix` сценария `setup.py`. На-

пример, команда `python setup.py install --prefix=/home/beazley/pypackages` установит модуль в каталог `/home/beazley/pypackages`. Когда установка выполняется в нестандартный каталог, наверняка может потребоваться изменить значение параметра настройки `sys.path`, чтобы интерпретатор смог отыскать вновь установленные модули.

Следует помнить, что многие расширения Python написаны на языке C или C++. Если вы загрузили дистрибутив с исходными текстами, то чтобы выполнить установку такого расширения, в вашей системе должен быть установлен компилятор C++. В UNIX, Linux и OS X это требование обычно не вызывает проблем. В Windows для этого традиционно требуется установить версию Microsoft Visual Studio. Если вы пользуетесь Windows, вероятно, будет лучше, если вы попробуете отыскать уже скомпилированную версию расширения.

Если у вас было установлено расширение `setuptools`, для установки пакетов можно использовать сценарий `easy_install`. Чтобы установить какой-то конкретный пакет, достаточно ввести команду `easy_install` *`имя_пакета`*. Если `setuptools` имеет корректные настройки, сценарий загрузит требуемый пакет из каталога PyPI, вместе со всеми зависимостями, и установит его. Конечно, время установки при этом может изменяться.

Если у вас появится желание поместить свои разработки в каталог PyPI, просто введите команду `python setup.py register`. Она выгрузит метаданные о последней версии вашего программного продукта в каталог (обратите внимание, что перед этим вы должны зарегистрироваться на сайте и получить имя пользователя и пароль).

9

Ввод и вывод

В этой главе рассматриваются основы ввода-вывода данных на языке Python, включая параметры командной строки, переменные окружения, файлы, поддержку Юникода и сериализацию объектов с помощью модуля pickle.

Чтение параметров командной строки

При запуске программы на языке Python параметры командной строки сохраняются в списке sys.argv. Первый элемент этого списка – имя программы. Остальные элементы – это параметры, указанные в командной строке *после* имени программы. Ниже приводится пример прототипа программы, которая вручную обрабатывает простые параметры командной строки:

```
import sys
if len(sys.argv) != 3:
    sys.stderr.write("Usage: python %s inputfile outputfile\n" % sys.argv[0])
    raise SystemExit(1)
inputfile = sys.argv[1]
outputfile = sys.argv[2]
```

Элемент sys.argv[0] в этой программе содержит имя сценария, выполняемого в данный момент. Запись сообщения об ошибке в sys.stderr и возбуждение исключения SystemExit с ненулевым кодом завершения, как показано в примере, – это стандартный прием вывода сообщений об ошибках в инструментах командной строки.

В простых случаях параметры командной строки можно обрабатывать вручную, однако сложную обработку параметров командной строки лучше производить с помощью модуля optparse. Например:

```
import optparse
p = optparse.OptionParser()

# Параметр имеет дополнительный аргумент
p.add_option("-o",action="store",dest="outfile")
```

```
p.add_option("--output",action="store",dest="outfile")

# Параметр устанавливает логический флаг
p.add_option("-d",action="store_true",dest="debug")
p.add_option("--debug",action="store_true",dest="debug")

# Установить значения по умолчанию для отдельных параметров
p.set_defaults(debug=False)

# Анализ командной строки
opts, args = p.parse_args()

# Извлечение значений параметров
outfile   = opts.outfile
debugmode = opts.debug
```

В этом примере было добавлено два типа параметров. Первый параметр `-o`, или `--output`, принимает обязательный аргумент. Эта особенность определяется аргументом `action='store'` в вызове метода `p.add_option()`. Второй параметр `-d`, или `--debug`, просто устанавливает логический флаг. Это определяется аргументом `action='store_true'` в вызове `p.add_option()`. Аргумент `dest` в вызове `p.add_option()` определяет имя атрибута, в котором будет храниться значение параметра после анализа. Метод `p.set_defaults()` устанавливает значения по умолчанию для одного или более параметров. Имена аргументов в вызове этого метода должны совпадать с именами атрибутов, в которых будут сохраняться значения параметров. Если значение по умолчанию не определено, атрибуты будут получать значение `None`.

Предыдущая программа правильно распознает все следующие параметры командной строки:

```
% python prog.py -o outfile -d infile1 ... infileN
% python prog.py --output=outfile --debug infile1 ... infileN
% python prog.py -h
% python prog.py --help
```

Анализ выполняется с помощью метода `p.parse_args()`. Этот метод возвращает кортеж из двух элементов (`opts, args`), где элемент `opts` — это объект, содержащий значения параметров, а `args` — это список элементов командной строки, которые не были опознаны как допустимые параметры. Значения параметров извлекаются из атрибутов `opts.`*`dest`*, где *`dest`* — это имя аргумента, указанного при добавлении параметра методом `p.add_option()`. Например, аргумент параметра `-o` или `--output` сохраняется в атрибуте `opts.outfile`, тогда как `args` — это список остальных аргументов, такой как `['infile1', ..., 'infileN']`. Модуль `optparse` автоматически добавляет параметр `-h`, или `--help`, в котором перечислены все допустимые параметры, которые должны вводиться пользователем. Кроме того, передача недопустимых параметров приводит к выводу сообщения об ошибке.

Этот пример демонстрирует простейший случай использования модуля `optparse`. Подробнее о некоторых дополнительных параметрах рассказывается в главе 19 «Службы операционной системы».

Переменные окружения

Доступ к переменным окружения осуществляется с помощью словаря `os.environ`. Например:

```
import os
path   = os.environ["PATH"]
user   = os.environ["USER"]
editor = os.environ["EDITOR"]
... и т.д. ...
```

Чтобы изменить переменную окружения, достаточно присвоить значение элементу словаря `os.environ`. Например:

```
os.environ["FOO"] = "BAR"
```

Изменения в словаре `os.environ` оказывают воздействие как на саму программу, так и на все подпроцессы, созданные интерпретатором Python.

Файлы и объекты файлов

Встроенная функция `open(`*`name`* `[,`*`mode`* `[,`*`bufsize`*`]])` открывает файл и создает объект файла, как показано ниже:

```
f = open("foo")     # Откроет "foo" для чтения
f = open("foo",'r') # Откроет "foo" для чтения (как и выше)
f = open("foo",'w') # Откроет "foo" для записи
```

Файл может быть открыт в режиме `'r'` – для чтения, в режиме `'w'` – для записи и в режиме `'a'` – для добавления в конец. Эти режимы предполагают выполнение операций с текстовыми файлами и могут неявно производить преобразование символа перевода строки `'\n'`. Например, в Windows при записи символа `'\n'` в действительности выводится последовательность из двух символов `'\r\n'` (а при чтении последовательность `'\r\n'` преобразуется обратно в одиночный символ `'\n'`). При работе с двоичными данными к символу режима открытия файла следует добавлять символ `'b'`, например: `'rb'` или `'wb'`. Этот символ отключает преобразование символа перевода строки и обязательно должен указываться для обеспечения переносимости программного кода, обрабатывающего двоичные данные (программисты, работающие в UNIX, часто допускают ошибку, опуская символ `'b'`, из-за отсутствия различий между текстовыми и двоичными файлами). Кроме того, из-за имеющихся различий в режимах часто можно увидеть такое определение текстового режима, как `'rt'`, `'wt'` или `'at'`, что более четко выражает намерения программиста.

Файл может быть открыт в режиме обновления, за счет использования символа (+), например: `'r+'` или `'w+'`. Когда файл открыт в режиме для обновления, допускается выполнять обе операции, чтения и записи, при условии, что все операции вывода будут выталкивать свои данные в файл до того, как будет выполнена операция ввода.

Если файл открывается в режиме `'w+'`, в момент открытия его длина усекается до нуля. Если файл открывается в режиме `'U'` или `'rU'`, включается

поддержка универсального символа перевода строки для режима чтения. Эта особенность упрощает обработку различных символов перевода строки (таких как '\n', '\r' и '\r\n'), используемых на разных платформах, преобразуя их в стандартный символ '\n' в строках, возвращаемых различными функциями ввода-вывода. Это может быть полезно, например, при разработке сценариев для системы UNIX, которые должны обрабатывать текстовые файлы, созданные программами в Windows.

Необязательный аргумент *bufsize* управляет буферизацией файла, где значение 0 означает отсутствие буферизации, значение 1 – построчную буферизацию и отрицательное значение – буферизацию в соответствии с параметрами системы по умолчанию. Любое другое положительное число интерпретируется как примерный размер буфера в байтах.

В версии Python 3 в функцию `open()` было добавлено четыре дополнительных аргумента – она вызывается как `open(`*name* `[,`*mode* `[,`*bufsize* `[, `*encoding* `[, `*errors* `[, `*newline* `[, `*closefd*`]]]]]]`). Аргумент *encoding* определяет имя кодировки символов, например: 'utf-8' или 'ascii'. Аргумент *errors* определяет политику обработки ошибок, связанных с кодировкой символов (дополнительная информация о Юникоде приводится в следующих разделах этой главы). Аргумент *newline* определяет порядок работы в режиме поддержки универсального символа перевода строки и может принимать значения `None`, '', '\n', '\r' или '\r\n'. Если имеет значение `None`, любые символы окончания строки, такие как '\n', '\r' или '\r\n', преобразуются в '\n'. Если имеет значение '' (пустая строка), любые символы окончания строки распознаются, как символы перевода строки, но в исходном тексте остаются в первоначальном виде. Если аргумент *newline* имеет любое другое допустимое значение, оно будет использоваться для преобразования символов окончания строки. Аргумент *closefd* определяет, должен ли в действительности закрываться дескриптор файла при вызове метода `close()`. По умолчанию имеет значение `True`. В табл. 9.1 перечислены методы, поддерживаемые объектами `file`.

Таблица 9.1. Методы файлов

Метод	Описание
f.`read([`*n*`])`	Читает из файла до *n* байтов.
f.`readline([`*n*`])`	Читает одну строку, но не более *n* байтов. Если аргумент `n` опущен, читает строку целиком.
f.`readlines([`*size*`])`	Читает все строки и возвращает список. С помощью необязательного аргумента *size* можно определить максимальное количество символов для чтения.
f.`write(`*s*`)`	Записывает строку *s*.
f.`writelines(`*lines*`)`	Записывает все строки из последовательности *lines*.
f.`close()`	Закрывает файл.
f.`tell()`	Возвращает текущую позицию в файле.
f.`seek(`*offset* `[, `*whence*`])`	Перемещает текущую позицию в новое место.

Метод	Описание
f.isatty()	Возвращает 1, если f представляет интерактивный терминал.
f.flush()	Выталкивает буферы вывода.
f.truncate([*size*])	Усекает размер файла до *size* байтов.
f.fileno()	Возвращает целочисленный дескриптор файла.
f.next()	Возвращает следующую строку или возбуждает исключение StopIteration. В Python 3 этот метод называется *f*.__next__().

Метод read() возвращает содержимое файла целиком, в виде одной строки, если не было указано значение необязательного аргумента, определяющее максимальное количество символов. Метод readline() возвращает следующую строку из файла, включая завершающий символ перевода строки; метод readlines() возвращает все строки из файла в виде списка строк. Метод readline() принимает необязательный аргумент, определяющий максимальную длину строки, *n*. Если строка длиннее, чем значение *n*, возвращаются первые *n* символов. Оставшиеся символы не уничтожаются и будут возвращены следующими операциями чтения. Метод readlines() принимает аргумент *size*, определяющий примерное число символов, которые должны быть прочитаны. Фактическое число прочитанных символов может оказаться больше этого значения, в зависимости от того, какой объем данных был буферизован.

Оба метода, readline() и readlines(), учитывают особенности операционной системы и корректно обрабатывают различные представления окончания строки (например, '\n' и '\r\n'). Если файл был открыт в режиме поддержки универсального символа перевода строки ('U' или 'rU'), символы окончания строки преобразуются в '\n'.

Методы read() и readline() сообщают о достижении конца файла, возвращая пустую строку. В следующем фрагменте показано, как использовать эту особенность, чтобы определить достижение конца файла:

```
while True:
    line = f.readline()
    if not line:          # Конец файла
        break
```

Файлы обеспечивают удобную возможность чтения всех строк в цикле for. Например:

```
for line in f:     # Итерации по всем строкам в файле
    # Выполнить некоторые действия со строкой
    ...
```

Следует знать, что в Python 2 различные операции чтения всегда возвращают 8-битные строки, независимо от выбранного режима доступа к файлу (текстовый или двоичный). В Python 3 эти операции возвращают стро-

ки Юникода, если файл был открыт в текстовом режиме, и строки байтов – если файл был открыт в двоичном режиме.

Метод `write()` записывает в файл строку, а метод `writelines()` – список строк. Методы `write()` и `writelines()` не добавляют символы перевода строки, поэтому выводимые строки должны включать все необходимые символы форматирования. Эти методы могут записывать в файл простые строки байтов, но только если файл был открыт в двоичном режиме.

Каждый объект файла внутри хранит файловый указатель, то есть смещение от начала файла в байтах, откуда начнется следующая операция чтения или записи. Метод `tell()` возвращает текущее значение указателя в виде длинного целого числа. Метод `seek()` используется для организации произвольного доступа к различным частям файла, принимая значение *`offset`* смещения и флаг *`whence`*, определяющий, относительно какой позиции в файле будет откладываться указанное смещение. Если *`whence`* имеет значение `0` (по умолчанию), метод `seek()` отложит смещение *`offset`* от начала файла; если *`whence`* имеет значение `1`, смещение будет отложено относительно текущей позиции в файле; а если *`whence`* имеет значение `2`, смещение будет отложено от конца файла. Метод `seek()` возвращает новое значение файлового указателя в виде целого числа. Следует отметить, что файловый указатель связан с объектом файла, который возвращается функцией `open()`, а не с самим файлом. Один и тот же файл можно открыть несколько раз в одной и той же программе (или в разных программах). При этом каждый экземпляр открытого файла будет иметь собственный, независимый от других, файловый указатель.

Метод `fileno()` возвращает целое число, представляющее дескриптор файла, который иногда может использоваться в низкоуровневых операциях ввода-вывода, доступных в некоторых библиотечных модулях. Например, модуль `fcntl` использует файловые дескрипторы для выполнения низкоуровневых операций управления файлами в системах UNIX.

Кроме того, объекты файлов имеют атрибуты, доступные только для чтения, перечисленные в табл. 9.2.

Таблица 9.2. Атрибуты объектов файлов

Атрибут	Описание
f.`closed`	Логическое значение, соответствующее состоянию файла: `False` – файл открыт, `True` – закрыт.
f.`mode`	Режим ввода-вывода.
f.`name`	Имя файла, если он создан с помощью функции `open()`. В противном случае это будет строка с именем существующего файла.
f.`softspace`	Логическое значение, сообщающее, должна ли инструкция `print` выводить пробел перед очередным значением. Классы, имитирующие файлы, должны предоставлять атрибут с этим именем, доступный для записи, инициализируемый нулем (только в Python 2).

Атрибут	Описание
f.newlines	Когда файл открывается в режиме поддержки универсального символа перевода строки, этот атрибут будет содержать представление символа окончания строки, фактически используемого в файле. Значением атрибута может быть None, которое говорит о том, что никаких символов завершения строки не было встречено; строка, содержащая '\n', '\r' или '\r\n', или кортеж со всеми встреченными символами завершения строки.
f.encoding	Строка с названием кодировки файла, если определена (например, 'latin-1' или 'utf-8'). Если кодировка не используется, этот атрибут имеет значение None.

Стандартный ввод, вывод и вывод сообщений об ошибках

Интерпретатор предоставляет три стандартных объекта файлов, известных, как *стандартный ввод*, *стандартный вывод* и *стандартный вывод сообщений об ошибках*, которые объявлены в модуле sys и доступны как sys.stdin, sys.stdout и sys.stderr соответственно. stdin — это объект файла, соответствующий потоку входных символов, передаваемых интерпретатору. stdout — это объект файла, который принимает данные от инструкции print. stderr — это объект файла, который принимает сообщения об ошибках. Практически всегда объект stdin отображается на клавиатуру, а объекты stdout и stderr воспроизводят текст на экране.

Методы, описанные в предыдущем разделе, могут применяться пользователем для выполнения простых операций ввода-вывода. Например, следующий фрагмент выводит данные на стандартный вывод и читает строку со стандартного ввода:

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

Дополнительно существует встроенная функция raw_input(*prompt*), которая читает строку текста из файла sys.stdin и может выводить приглашение к вводу:

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

Строки, возвращаемые функцией raw_input(), не включают завершающий символ перевода строки. Этим она отличается от простых операций чтения из файла sys.stdin, которые включают перевод строки в возвращаемый текст. В Python 3 функция raw_input() была переименована в input().

Ввод символа прерывания работы программы с клавиатуры (обычно генерируется комбинацией клавиш Ctrl+C) приводит к возбуждению исключения KeyboardInterrupt, которое можно перехватить с помощью обработчика исключений.

В случае необходимости значения переменных sys.stdout, sys.stdin и sys.stderr могут замещаться другими объектами файлов, в этом случае инструкция print и функции ввода будут использовать новые значения. Если позднее потребуется восстановить оригинальное значение переменной sys.stdout, его предварительно следует сохранить. Кроме того, оригинальные значения переменных sys.stdout, sys.stdin и sys.stderr, которые присваиваются при запуске интерпретатора, доступны в переменных sys.__stdout__, sys.__stdin__ и sys.__stderr__ соответственно.

Обратите внимание, что в некоторых случаях интегрированные среды разработки могут подменять значения переменных sys.stdin, sys.stdout и sys.stderr. Например, когда интерпретатор Python запускается под управлением IDLE, в переменную sys.stdin записывается ссылка на объект, который своим поведением напоминает файл, но в действительности является объектом, созданным средой разработки. В этом случае некоторые низкоуровневые методы, такие как read() и seek(), могут оказаться недоступными.

Инструкция print

Для вывода в файл, на который указывает переменная sys.stdout, в Python 2 используется специальная инструкция print. Инструкция print принимает список объектов, разделенных запятыми, например:

```
print “Значения: “, x, y, z
```

Для каждого объекта вызывается функция str(), которая генерирует строковое представление этого объекта. Затем эти строки объединяются в заключительную строку, где они разделяются символом пробела. Полученная строка завершается символом перевода строки, если в инструкции print отсутствует завершающая запятая. В противном случае следующая инструкция print добавит ведущий пробел перед выводом других объектов. Выводом этого пробела управляет атрибут softspace файла, в который производится вывод.

```
print “Значения: “, x, y, z, w
# Вывести тот же текст с помощью двух инструкций print
print “Значения: “, x, y,  # Завершающий символ перевода строки не выводится
print z, w                 # Перед значением z будет выведен пробел
```

Для вывода форматированных строк используется оператор форматирования (%) или метод .format(), как было описано в главе 4 «Операторы и выражения». Например:

```
print “Значения: %d %7.5f %s” % (x,y,z) # Форматированный ввод-вывод
print “Значения: {0:d} {1:7.5f} {2}”.format(x,y,z)
```

Файл, куда производится вывод инструкцией print, можно заменить, добавив специальный модификатор *>>file*, со следующей за ним запятой, где *file* – это объект файла, открытый в режиме для записи. Например:

```
f = open(“output”,”w”)
print >>f, “привет, мир”
...
f.close()
```

Функция print()

Одним из самых существенных изменений в Python 3 является преобразование инструкции `print` в функцию. В Python 2.6 инструкция `print` может вызываться, как функция, если добавить инструкцию `from __future__ import print_function` в каждый модуль, который ее использует. Функция `print()` действует практически так же, как и инструкция `print`, описанная в предыдущем разделе.

Чтобы вывести несколько значений, достаточно просто перечислить их через запятую в списке аргументов `print()`, например:

```
print("Значения: ", x, y, z)
```

Подавить или изменить символ окончания строки можно с помощью именованного аргумента `end=`*`ending`*. Например:

```
print("Значения: ", x, y, z, end='') # Подавит вывод конца строки
```

Перенаправить вывод в файл можно с помощью именованного аргумента `file=`*`outfile`*. Например:

```
print("Значения: ", x, y, z, file=f) # Произведет вывод в объект файла f
```

Изменить символ-разделитель между элементами можно с помощью именованного аргумента `sep=`*`sepchr`*. Например:

```
print("Значения: ", x, y, z, sep=',') # Выведет запятую между значениями
```

Интерполяция переменных при выводе текста

Типичная проблема, возникающая при выводе, связана с вводом больших фрагментов текста, содержащих переменные. Во многих языках сценариев, таких как Perl и PHP, имеется возможность вставлять имена переменных в строки, используя оператор ($) подстановки значений переменных (то есть `$name`, `$address` и так далее). В языке Python отсутствует прямой эквивалент этой возможности, но ее можно имитировать, используя операцию форматирования строк в соединении со строками в тройных кавычках. Например, можно было бы создать шаблон письма, в котором присутствуют переменные элементы `name`, `item` и `amount`, как показано ниже:

```
# Обратите внимание: завершающий символ обратного слэша, сразу за """
# подавляет вывод первой пустой строки
form = """\
Уважаемый  %(name)s,
Пожалуйста, верните обратно %(item)s или заплатите $%(amount)0.2f.
                                                  Искренне ваш,

                                                Joe Python User
"""
print form % { 'name': 'Мистер Буш',
               'item': 'блендер',
               'amount': 50.00,
             }
```

В результате будет получен следующий вывод:

```
Уважаемый Мистер Буш,

Пожалуйста, верните обратно блендер или заплатите $50.00.

                                                    Искренне ваш,
                                                  Joe Python User
```

Метод `format()` является более современной альтернативой, позволяя сделать предыдущий программный код более наглядным. Например:

```
form = """\
Уважаемый  {name},
Пожалуйста, верните обратно {item} или заплатите {amount:0.2f}.
                                                    Искренне ваш,

                                                  Joe Python User
"""
print form.format(name='Мистер Буш', item='блендер', amount=50.0)
```

В некоторых случаях можно также использовать метод `Template()` из модуля `string`, как показано ниже:

```
import string
form = string.Template("""\
Уважаемый  $name,
Пожалуйста, верните обратно $item или заплатите $amount.
                                                    Искренне ваш,

                                                  Joe Python User
""")
print form.substitute({'name': 'Мистер Буш',
                       'item': 'блендер',
                       'amount': "%0.2f" % 50.0})
```

В данном случае специальные переменные, начинающиеся с символа `$`, в строке являются символами подстановки. Метод `form.substitute()` принимает словарь подставляемых значений и возвращает новую строку. Несмотря на простоту предыдущего примера, этот подход не всегда является достаточно мощным решением при создании текста. Веб-фреймворки, как и многие другие фреймворки разработки приложений, обычно предоставляют свои собственные механизмы шаблонов, которые поддерживают встроенные операторы управления потоком выполнения, подстановки переменных, включения файлов и другие расширенные особенности.

Вывод с помощью генераторов

Непосредственная работа с файлами – это самая знакомая программистам модель ввода-вывода. Однако для вывода последовательности фрагментов данных также могут использоваться функции-генераторы. Для этого достаточно просто задействовать инструкцию `yield` вместо метода `write()` или инструкции `print`. Например:

```
def countdown(n):
    while n > 0:
        yield "Очередное значение %d\n" % n
        n -= 1
    yield "Конец!\n"
```

Создание потоков выходных данных таким способом обеспечивает значительную гибкость, потому что создание выходного потока отделено от программного кода, который фактически направляет поток адресату. Например, при желании вывод можно направить в файл *f*, как показано ниже:

```
count = countdown(5)
f.writelines(count)
```

Точно так же можно направить данные в сокет *s*, например:

```
for chunk in count:
    s.sendall(chunk)
```

Или просто записать данные в строку:

```
out = "".join(count)
```

Более сложные приложения могут использовать такой подход, реализовав свой собственный механизм буферизации ввода-вывода. Например, генератор мог бы производить небольшие фрагменты текста, а другая функция могла бы собирать эти фрагменты в большие буферы, повышая тем самым эффективность операций ввода-вывода:

```
chunks = []
buffered_size = 0
for chunk in count:
    chunks.append(chunk)
    buffered_size += len(chunk)
    if buffered_size >= MAXBUFFERSIZE:
        outf.write("".join(chunks))
        chunks.clear()
        buffered_size = 0
outf.write("".join(chunks)
```

Для программ, направляющих вывод в файлы или в сетевые соединения, подход на основе генераторов может также способствовать существенному снижению требований к объему памяти, потому что весь выходной поток зачастую может генерироваться и обрабатываться небольшими фрагментами, в противоположность методу, когда сначала данные собираются в одну большую строку или список строк. Такой способ вывода данных иногда можно встретить в программах на языке Python, взаимодействующих со шлюзовым интерфейсом веб-служб (Web Services Gateway Interface – WSGI), который используется для обмена информацией между программными компонентами в некоторых веб-фреймворках.

Обработка строк Юникода

Типичной проблемой, связанной с вводом-выводом, является обработка интернациональных символов, представленных символами Юникода.

Если имеется некоторая строка *s*, состоящая из байтов, которые являются кодированным представлением символов Юникода, для ее преобразования в соответствующую строку Юникода можно использовать метод *s*.decode([*encoding* [,*errors*]]). Преобразовать строку Юникода *u* в кодированную строку байтов можно с помощью строкового метода *u*.encode([*encoding* [, *errors*]]). Оба эти метода, выполняющие преобразование, принимают специальное имя кодировки, определяющей порядок отображения символов Юникода в последовательности 8-битных символов и наоборот. Аргумент *encoding* определяется как строка и может быть именем одной из более чем сотни различных названий кодировок. Ниже приводятся наиболее часто встречаемые значения:

Значение	Описание
'ascii'	7-битная кодировка ASCII
'latin-1' или 'iso-8859-1'	ISO 8859-1 Latin-1
'cp1252'	Кодировка Windows 1252
'utf-8'	8-битная кодировка с символами переменной длины
'utf-16'	16-битная кодировка с символами переменной длины (может быть с прямым и с обратным порядком следования байтов)
'utf-16-le'	UTF-16, кодировка с обратным порядком (little endian) следования байтов
'utf-16-be'	UTF-16, кодировка с прямым порядком (big endian) следования байтов
'unicode-escape'	Формат литералов строк Юникода u"string"
'raw-unicode-escape'	Формат литералов «сырых» строк Юникода ur"string"

Кодировку по умолчанию, которая устанавливается модулем site, можно получить с помощью функции sys.getdefaultencoding(). Во многих случаях по умолчанию используется кодировка 'ascii', которая напрямую отображает символы ASCII с кодами в диапазоне [0x00,0x7f] в символы Юникода в диапазоне [U+0000, U+007F]. Однако кодировка 'utf-8' также используется достаточно часто. Технические подробности, касающиеся наиболее распространенных кодировок, приводятся в следующем разделе.

Когда используется метод *s*.decode(), всегда предполагается, что *s* – это строка байтов. В Python 2 *s* интерпретируется, как стандартная строка, но в Python 3 строка *s* должна быть объектом специального типа bytes. Аналогично результатом вызова метода *t*.encode() всегда является последовательность байтов. Если вас волнует проблема переносимости, замечу, что в Python 2 эти методы могут стать источником путаницы. Например, строки в Python 2 обладают обоими методами, decode() и encode(), тогда как в Python 3 строки имеют только метод encode(), а тип bytes – только метод

decode(). Чтобы упростить последующий перенос программного кода для Python 2, используйте метод encode() только со строками Юникода, а метод decode() — только со строками байтов.

Если при преобразовании строковых значений встретится символ, который не может быть преобразован в требуемую кодировку, будет возбуждено исключение UnicodeError. Например, если попытаться преобразовать в кодировку 'ascii' строку, содержащую символы Юникода, такие как U+1F28, возникнет ошибка кодирования, так как значение этого символа окажется слишком большим для представления в наборе символов ASCII. Аргумент *errors* методов encode() и decode() определяет порядок обработки ошибок, возникающих при преобразовании. Это должна быть строка с одним из следующих значений:

Значение	Описание
'strict'	В случае появления ошибки кодирования и декодирования возбуждается исключение UnicodeError.
'ignore'	Недопустимые символы игнорируются.
'replace'	Замещает недопустимые символы символом замены (U+FFFD — в строках Юникода, '?' — в стандартных строках).
'backslashreplace'	Замещает недопустимые символы соответствующими им экранированными последовательностями, принятыми в языке Python. Например, символ U+1234 будет замещен последовательностью '\u1234'.
'xmlcharrefreplace'	Замещает недопустимые символы ссылками в формате XML. Например, символ U+1234 будет замещен последовательностью 'ሴ'.

По умолчанию используется значение 'strict'.

Политика обработки ошибок 'xmlcharrefreplace' часто является удобным способом встраивания интернациональных символов в текст ASCII вебстраниц. Например, если вывести строку Юникода 'Jalape\u00f1o', закодировав ее в кодировку ASCII с политикой обработки ошибок 'xmlcharrefreplace', броузеры практически всегда корректно отобразят текст строки как "Jalapeco", а не как немного искаженную альтернативу.

Чтобы избавить себя от лишней головной боли, никогда не смешивайте в выражениях кодированные строки байтов и декодированные строки (например, в операции + конкатенации). В Python 3 это вообще невозможно, но в Python 2 такие операции будут выполняться без вывода каких-либо сообщений об ошибках, автоматически преобразуя строки байтов в строки Юникода, используя значение кодировки по умолчанию. Такое поведение часто приводит к неожиданным результатам или непонятным сообщениям об ошибках. Поэтому в программах всегда следует строго отделять кодированные и некодированные символьные данные.

Ввод-вывод Юникода

При работе со строками Юникода нет никакой возможности напрямую записать данные в файл. Это обусловлено тем, что во внутреннем представлении символы Юникода являются многобайтовыми целыми числами и при непосредственной записи таких чисел в поток вывода возникают проблемы, связанные с порядком следования байтов. Например, вам пришлось бы решить, записывать ли символ Юникода `U+HHLL` в формате с «обратным порядком следования байтов», как `LL HH`, или в формате с «прямым порядком следования байтов», как `HH LL`. Кроме того, другие инструменты, выполняющие обработку текста Юникода, должны знать, какая кодировка была использована при записи.

Из-за этой проблемы преобразование строк Юникода во внешнее представление всегда выполняется в соответствии с определенными правилами кодирования, которые четко определяют, как преобразовывать символы Юникода в последовательности байтов. По этой причине, чтобы обеспечить поддержку ввода-вывода Юникода, понятия кодирования и декодирования, описанные в предыдущем разделе, были распространены на файлы. Встроенный модуль `codecs` содержит набор функций, позволяющих выполнять преобразование байтовых данных в строки Юникода и обратно, в соответствии с различными схемами кодирования.

Самый прямолинейный, пожалуй, способ обработки файлов Юникода заключается в использовании функции `codecs.open(`*`filename`* `[, `*`mode`* `[, `*`encoding`* `[, `*`errors`*`]]])`, как показано ниже:

```
f = codecs.open('foo.txt','r','utf-8','strict') # Чтение
g = codecs.open('bar.txt','w','utf-8')          # Запись
```

Здесь создаются объекты файлов, из одного производится чтение строки Юникода, в другой – запись. Аргумент *`encoding`* определяет способ кодирования символов, который будет использоваться при чтении и записи данных. Аргумент *`errors`* определяет политику обработки ошибок и может принимать одно из значений: `'strict'`, `'ignore'`, `'replace'`, `'backslashreplace'` или `'xmlcharrefreplace'`, как было описано в предыдущем разделе.

Если в программе уже имеется объект файла, то функцию `codecs.EncodedFile(`*`file`*`, `*`inputenc`* `[, `*`outputenc`* `[, `*`errors`*`]])` можно использовать в качестве обертки вокруг этого объекта, выполняющей кодирование. Например:

```
f = open("foo.txt","rb")
...
fenc = codecs.EncodedFile(f,'utf-8')
```

В данном примере интерпретация данных, читаемых из файла, будет выполняться в соответствии с кодировкой, указанной в аргументе *`inputenc`*, а при записи – в соответствии с кодировкой в аргументе *`outputenc`*. Если аргумент *`outputenc`* опущен, по умолчанию будет использоваться значение аргумента *`inputenc`*. Аргумент *`errors`* имеет то же назначение, как было описано выше. При использовании обертки `EncodedFile` вокруг существующего объекта файла файл должен быть открыт в двоичном режиме. В противном

случае преобразование символов перевода строки может препятствовать кодированию.

При работе с файлами Юникода помните, что информация об используемой кодировке может включаться в сам файл. Например, парсеры разметки XML **могут определить кодировку документа, просматривая первые несколько байтов** строки `'<?xml ...>'`. Если первые четыре байта в этой строке —`3C 3F 78 6D` (`'<?xm'`), можно предположить, что используется кодировка UTF-8. Если первые четыре байта — `00 3C 00 3F` или `3C 00 3F 00`, можно предположить, что используется кодировка **UTF-16 с прямым порядком следования байтов** или **UTF-16 с обратным порядком следования байтов** соответственно. Кроме того, кодировка документа может указываться в заголовках MIME или в виде других атрибутов элементов документа. Например:

```
<?xml ... encoding="ISO-8859-1" ... ?>
```

Аналогично файлы Юникода также могут включать специальные маркеры порядка следования байтов (Byte-Order Markers — BOM), указывающие на свойства кодировки символов. Для этих целей зарезервирован символ Юникода `U+FEFF`. **Обычно маркер записывается как первый символ в файле.** Программы, прочитав этот символ, могут по следующим за ним байтам определить используемую кодировку (например, `'\xff\xfe'` — для UTF-16-LE или `'\xfe\xff'` — для UTF-16-BE). После определения кодировки символ BOM **отбрасывается и обрабатывается остальная часть файла. К сожалению,** все эти дополнительные манипуляции с маркерами **BOM не выполняются** внутренней реализацией. Чаще всего вам придется самим позаботиться об этом, если приложение должно гарантировать такую возможность.

Когда кодировка определяется подобным способом, по первым байтам в документе, можно использовать следующий ниже программный код, превращающий входной файл в поток кодированных данных:

```
f = open("somefile","rb")
# Определить кодировку данных
...
# Обернуть объект файла, указав требуемую кодировку.
# Далее предполагается, что символ BOM (если таковой имеется)
# уже был отброшен предыдущими инструкциями.
fenc = codecs.EncodedFile(f,encoding)
data = fenc.read()
```

Кодировки данных Юникода

В табл. 9.3 перечислены кодировки в модуле `codecs`, используемые наиболее часто.

Таблица 9.3. Кодировки, реализованные в модуле codecs

Кодировка	Описание
`'ascii'`	Кодировка ASCII
`'latin-1'` **или** `'iso-8859-1'`	Кодировка Latin-1 или ISO 8859-1

Таблица 9.3 (продолжение)

'cp437'	Кодировка CP437
'cp1252'	Кодировка CP1252
'utf-8'	8-битная кодировка с символами переменной длины
'utf-16'	16-битная кодировка с символами переменной длины
'utf-16-le'	Кодировка UTF-16, но с явно указанным обратным порядком (little endian) следования байтов
'utf-16-be'	Кодировка UTF-16, но с явно указанным прямым порядком (big endian) следования байтов
'unicode-escape'	Формат литералов строк Юникода u"*string*"
'raw-unicode-escape'	Формат литералов «сырых» строк Юникода ur"*string*"

В следующих разделах приводятся более подробные описания каждой из кодировок.

Кодировка 'ascii'

В кодировке 'ascii' значения символов ограничены диапазонами [0x00,0x7f] и [U+0000, U+007F]. Любые символы, со значениями за пределами этих диапазонов, считаются недопустимыми.

Кодировка 'iso-8859-1', 'latin-1'

Символы могут иметь любые 8-битные значения в диапазонах [0x00,0xff] и [U+0000, U+00FF]. Символы со значениями в диапазоне [0x00,0x7f] соответствуют символам ASCII. Символы со значениями в диапазоне [0x80,0xff] соответствуют символам в кодировке ISO-8859-1, или расширенному набору символов ASCII. Любые символы со значениями за пределами диапазона [0x00,0xff] вызывают появление ошибки.

Кодировка 'cp437'

Эта кодировка похожа на кодировку 'iso-8859-1' и используется интерпретатором Python по умолчанию, когда программа запускается в Windows как консольное приложение. Некоторые символы, со значениями в диапазоне [x80,0xff], интерпретируются как специальные символы, используемые при отображении меню, окон и рамок в устаревших программах, написанных для DOS.

Кодировка 'cp1252'

Эта кодировка очень похожа на кодировку 'iso-8859-1' и используется в Windows. Однако эта кодировка определяет символы в диапазоне [0x80-0x9f], которые не определены в кодировке 'iso-8859-1' и имеют другие кодовые пункты в Юникоде.

Кодировка 'utf-8'

UTF-8 – это кодировка, в которой символы могут иметь переменную длину, что позволяет представлять все символы Юникода. Символы ASCII в этой кодировке представлены единственным байтом, со значением в диапазоне 0–127. Все остальные символы представлены последовательностями байтов длиной от 2 до 3 байтов. Порядок кодирования этих байтов приводится ниже:

Символ Юникода	Байт 0	Байт 1	Байт 2
U+0000 - U+007F	0*nnnnnnn*		
U+007F - U+07FF	110*nnnnn*	10*nnnnnn*	
U+0800 - U+FFFF	1110*nnnn*	10*nnnnnn*	10*nnnnnn*

Первый байт в 2-байтовых последовательностях всегда начинается с последовательности битов 110. В 3-байтовых последовательностях первый байт начинается с последовательности битов 1110. Все последующие байты данных в многобайтовых последовательностях начинаются с последовательности битов 10.

Вообще говоря, кодировка UTF-8 позволяет использовать многобайтовые последовательности длиной до 6 байтов. Для кодирования пар символов Юникода в Python используются 4-байтовые последовательности UTF-8, которые называются *суррогатными парами*. Оба символа в таких парах имеют значения в диапазоне [U+D800, U+DFFF] и при объединении составляют 20-битный код символа. Суррогатное кодирование выполняется следующим образом: 4-байтовая последовательность 11110*nnn* 10*nnnnnn* 10*mmmm* 10*mmmmm* кодируется как пара U+D800 + N, U+DC00 + M, где N – это старшие 10 битов, а M – младшие 10 битов 20-битового символа, представленного в виде 4-байтовой последовательности **UTF-8. Пяти- и шестибайтовые последовательности UTF-8 (в первом байте обозначаются последовательностями битов** 111110 и 1111110 соответственно) используются для кодирования символов с 32-битными значениями. Эти значения не поддерживаются в языке Python **и в настоящее время, если они появляются в потоке данных, вызывают исключение** UnicodeError.

Кодировка **UTF-8 имеет множество полезных свойств, позволяющих использовать эту кодировку в устаревших приложениях. Во-первых,** стандартные символы ASCII представлены в ней своими собственными кодами. Это означает, что строки символов **ASCII в кодировке UTF-8 ничем не отличаются** от традиционных строк **ASCII. Во-вторых, кодировка UTF-8 не использует** байты со значением NULL в многобайтовых последовательностях. Благодаря этому существующие программы, опирающиеся на использование библиотеки языка **C, и программы, в которых используются 8-битные** строки, оканчивающиеся символом NULL, смогут работать со строками **UTF-8. Наконец, кодировка UTF-8 сохраняет возможность лексикографического** упорядочивания строк. То есть, если имеются строки Юникода a и b, где a < b, то после преобразования строк в кодировку UTF-8 условие

a < b также будет выполняться. Поэтому алгоритмы сортировки и упорядочения, написанные для 8-битных строк, также будут работать со строками в кодировке UTF-8.

Кодировки 'utf-16', 'utf-16-be' и 'utf-16-le'

UTF-16 – это 16-битная кодировка, в которой символы Юникода переменной длины записываются 16-битными значениями. Если порядок следования байтов не указан, предполагается кодирование с прямым порядком следования байтов (big endian). Кроме того, для явного указания порядка следования байтов в потоке данных UTF-16 может использоваться маркер U+FEFF. В кодировке с прямым порядком следования байтов значение U+FEFF представляет символ Юникода неразрывного пробела нулевой ширины, тогда как значение с переставленными байтами, U+FFFE, является недопустимым символом Юникода. Благодаря этому последовательности байтов FE FF или FF FE могут использоваться для определения порядка следования байтов в потоке данных. При чтении данных Юникода Python удаляет маркеры порядка следования байтов из окончательной строки Юникода.

Кодировка 'utf-16-be' явно определяет кодировку UTF-16 с прямым порядком следования байтов (big endian).

Кодировка 'utf-16-le' явно определяет кодировку UTF-16 с обратным порядком следования байтов (little endian).

Кодировка UTF-16 имеет расширения, обеспечивающие возможность поддержки символов со значениями больше 16 битов, однако в настоящее время ни одно из этих расширений не поддерживается.

Кодировки 'unicode-escape' и 'raw-unicode-escape'

Эти кодировки используются для преобразования строк Юникода в формат, который используется в языке Python для представления литералов строк Юникода и литералов «сырых» строк Юникода. Например:

```
s = u'\u14a8\u0345\u2a34'
t = s.encode('unicode-escape')     # t = '\u14a8\u0345\u2a34'
```

Свойства символов Юникода

В дополнение к операциям ввода-вывода в программах, где используется Юникод, может потребоваться проверить различные свойства символов Юникода, такие как регистр символа, принадлежность к цифрам и пробельным символам. Доступ к базе данных свойств символов Юникода обеспечивает модуль unicodedata. Вообще говоря, свойства символов можно получить вызовом функции unicodedata.category(*c*). Например, вызов unicodedata.category(u"A") вернет результат 'Lu', который говорит, что символ является заглавной буквой.

Еще одна проблема заключается в том, что одна и та же строка Юникода может иметь несколько представлений. Например, символ U+00F1 (ñ) может быть представлен единственным символом U+00F1 или многосимвольной последовательностью U+006e U+0303 (n, ˜). Если единообразие обработки

строк Юникода имеет важное значение, единство представления символов можно обеспечить с помощью функции unicodedata.normalize(). Например, вызов unicodedata.normalize('NFC', *s*) гарантирует, что все символы в строке *s* будут скомпонованы и ни один из них не будет представлен последовательностью комбинируемых символов.

Подробнее о базе данных символов Юникода и о модуле unicodedata рассказывается в главе 16 «Строки и обработка текста».

Сохранение объектов и модуль pickle

Очень часто бывает необходимо сохранить содержимое объекта в файле и восстановить его из файла. Чтобы решить эту проблему, можно написать пару функций, которые будут просто читать из файла и записывать данные в файл в специальном формате. Альтернативное решение заключается в использовании модулей pickle и shelve.

Модуль pickle преобразует объект в последовательность байтов, которая может быть записана в файл, а потом прочитана из файла. Интерфейс модуля pickle прост, он состоит из функций dump() и load(). Например, следующий фрагмент демонстрирует возможность записи объекта в файл:

```
import pickle
obj = SomeObject()
f = open(filename,'wb')
pickle.dump(obj, f)       # Сохранит объект в f
f.close()
```

Следующий фрагмент восстанавливает объект:

```
import pickle
f = open(filename,'rb')
obj = pickle.load(f)      # Восстановит объект
f.close()
```

Последовательность объектов можно сохранить серией вызовов функции dump(), один за другим. Восстановить эти объекты можно аналогичной серией вызовов функции load().

Модуль shelve похож на модуль pickle, но сохраняет объекты в базе данных, своей структурой напоминающей словарь:

```
import shelve
obj = SomeObject()
db = shelve.open("filename") # Открыть хранилище
db['key'] = obj              # Сохранить объект в хранилище
...
obj = db['key']              # Извлечь объект
db.close()                   # Закрыть хранилище
```

Хотя объект, создаваемый модулем shelve, выглядит как словарь, тем не менее он имеет некоторые ограничения. Во-первых, ключи могут быть только строками. Во-вторых, значения, сохраняемые в хранилище, должны быть совместимы с требованиями модуля pickle. Этим требованиям отвечает

большинство объектов Python, однако некоторые объекты специального назначения, такие как файлы и сетевые соединения, хранящие информацию о внутреннем состоянии, не могут быть сохранены и восстановлены таким способом.

Формат данных, который используется модулем `pickle`, характерен для языка Python. Однако этот формат несколько раз изменялся с выходом новых версий Python. Выбор протокола[1] может быть осуществлен с помощью дополнительного аргумента *protocol* функции `dump(`*obj, file, protocol*`)` в модуле `pickle`. По умолчанию используется протокол 0. Это самый старый формат представления данных модулем `pickle`, который может восприниматься практически всеми версиями Python. Однако этот формат несовместим со многими современными особенностями классов в языке Python, такими как слоты. В протоколах 1 и 2 используется более эффективное представление двоичных данных. Чтобы использовать эти альтернативные протоколы, необходимо выполнить такие операции, как:

```
import pickle
obj = SomeObject()
f = open(filename,'wb')
pickle.dump(obj,f,2)                          # Использовать протокол 2
pickle.dump(obj,f,pickle.HIGHEST_PROTOCOL) # Использовать самый современный
                                              # протокол
f.close()
```

При восстановлении объектов с помощью функции `load()` указывать протокол не требуется, так как номер протокола записывается непосредственно в файл.

Аналогично модуль `shelve` может создавать хранилища для сохранения объектов Python с использованием альтернативного протокола модуля `pickle`, например:

```
import shelve
db = shelve.open(filename,protocol=2)
...
```

При работе с пользовательскими объектами обычно не требуется выполнять дополнительные операции с привлечением модуля `pickle` или `shelve`. Однако объекты могут определять дополнительные методы `__getstate__()` и `__setstate__()` для оказания помощи процессу сохранения и восстановления. Метод `__getstate__()`, если он определен, вызывается для создания значения, являющегося представлением состояния объекта. Значение, возвращаемое методом `__getstate__()`, может быть строкой, кортежем, списком или словарем. Метод `__setstate__()` принимает это значение в процессе чтения объекта из файла и должен восстанавливать его состояние, исходя из этого значения. Ниже приводится пример, демонстрирующий реализации этих методов в объекте, который создает сетевое подключение. Несмотря на то что фактическое подключение нельзя сохранить в файле,

[1] Имеется в виду выбор номера версии формата. – *Прим. перев.*

в объекте имеется достаточный объем информации, чтобы возобновить это подключение после того, как объект будет восстановлен из файла:

```
import socket
class Client(object):
    def __init__(self,addr):
        self.server_addr = addr
        self.sock = socket.socket(socket.AF_INET,socket.SOCK_STREAM)
        self.sock.connect(addr)
    def __getstate__(self):
        return self.server_addr
    def __setstate__(self,value):
        self.server_addr = value
        self.sock = socket.socket(socket.AF_INET,socket.SOCK_STREAM)
        self.sock.connect(self.server_addr)
```

Поскольку модуль `pickle` использует формат, характерный только для языка Python, эту возможность нельзя использовать для обмена данными между приложениями, написанными на других языках программирования. Кроме того, из-за проблем безопасности программы не должны обрабатывать сохраненные данные, полученные из непроверенных источников (опытный злоумышленник сможет подделать файл с данными так, чтобы обеспечить выполнение любых системных команд в процессе его чтения).

Модули `pickle` и `shelve` имеют множество параметров настройки и способов использования. Более подробные сведения о них приводятся в главе 13 «Службы Python времени выполнения».

10

Среда выполнения

В этой главе описывается окружение, в котором выполняются программы на языке Python. **Целью главы является описание поведения интерпретатора** во время выполнения, включая запуск, настройку и завершение программы.

Параметры интерпретатора и окружение

Существует множество параметров, управляющих поведением интерпретатора во время выполнения и окружением. Параметры передаются интерпретатору с помощью командной строки, как показано ниже:

```
python [options] [-c cmd | filename | - ] [args]
```

В табл. 10.1 приводится перечень наиболее часто используемых параметров командной строки:

Таблица 10.1. Параметры командной строки, принимаемые интерпретатором

Параметр	Описание
`-3`	Включить вывод предупреждений об использовании особенностей, которые были удалены или изменены в Python 3.
`-B`	Не создавать файлы с расширениями `.pyc` и `.pyo` инструкцией `import`.
`-E`	Игнорировать переменные окружения.
`-h`	Вывести список всех доступных параметров командной строки.
`-i`	Перейти в интерактивный режим по завершении выполнения программы.
`-m module`	Запустить библиотечный модуль *module*, как самостоятельную программу.
`-O`	Включить режим оптимизации.

Параметр	Описание
-OO	Включить режим оптимизации и дополнительно удалить строки документирования при создании файла .pyo.
-Q *arg*	Определить поведение оператора деления в версии Python 2. Аргумент *arg* может быть одним из следующих: -Qold (по умолчанию), -Qnew, -Qwarn или -Qwarnall.
-s	Предотвратить возможность добавления пользователем новых каталогов в переменную sys.path.
-S	Не импортировать модуль site во время инициализации.
-t	Вывести предупреждение в случае непоследовательного использования символов табуляции.
-tt	Возбудить исключение TabError в случае непоследовательного использования символов табуляции.
-u	Отключить буферизацию для потоков stdout и stderr.
-U	Интерпретировать все строковые литералы как строки Юникода (только в Python 2).
-v	Включить режим вывода отладочной информации для инструкций import.
-V	Вывести номер версии и выйти.
-x	Пропустить первую строку в исходной программе.
-c *cmd*	Выполнить строку *cmd*, как инструкцию.

Параметр -i переводит интерпретатор в интерактивный режим сразу по завершении выполнения программы, что бывает очень удобно при отладке. Параметр -m запускает библиотечный модуль как сценарий, который выполняется под именем модуля __main__. Параметры -O и -OO включают режим оптимизации скомпилированных файлов с байт-кодом; описываются в главе 8 «Модули, пакеты и дистрибутивы». Параметр -S пропускает на этапе инициализации импортирование модуля site, который описывается в разделе «Файлы с настройками местоположения библиотек». Параметры -t, -tt и -v позволяют получать дополнительные предупреждения и отладочную информацию. Параметр -x заставляет интерпретатор пропустить первую строку программы и может использоваться в случае, когда эта строка не является допустимой инструкцией языка Python (например, когда первая строка в сценарии запускает интерпретатор Python).

Имя программы указывается после всех параметров интерпретатора. Если имя не указано или вместо имени файла используется символ дефиса (-), интерпретатор будет читать текст программы со стандартного ввода. Если устройством стандартного ввода является интерактивный терминал, будут выведены начальная информация и приглашение к вводу. В противном случае интерпретатор откроет указанный файл и будет выполнять инструкции в нем, пока не достигнет конца файла. Параметр -c *cmd* может использоваться для выполнения коротких программ в форме параметра командной строки, например: python -c "print('hello world')".

Параметры командной строки, находящиеся после имени программы или после дефиса (-), передаются программе в переменной sys.argv, как описывается в разделе «Чтение параметров командной строки» в главе 9 «Ввод и вывод».

Кроме того, интерпретатор читает значения переменных окружения, которые перечислены в табл. 10.2:

Таблица 10.2. Переменные окружения, используемые интерпретатором

Переменная окружения	Описание
PYTHONPATH	Список каталогов, разделенных двоеточием, образующий путь поиска модулей.
PYTHONSTARTUP	Файл, выполняемый при запуске интерпретатора в интерактивном режиме.
PYTHONHOME	Каталог установки Python.
PYTHONINSPECT	Имеет значение параметра -i.
PYTHONUNBUFFERED	Имеет значение параметра -u.
PYTHONIOENCODING	Кодировка по умолчанию и политика обработки ошибок при работе с потоками stdin, stdout и stderr. Значение этой переменной – строка вида "*encoding*[:*errors*]", например: "utf-8" или "utf-8:ignore".
PYTHONDONTWRITEBYTECODE	Имеет значение параметра -B.
PYTHONOPTIMIZE	Имеет значение параметра -O.
PYTHONNOUSERSITE	Имеет значение параметра -s.
PYTHONVERBOSE	Имеет значение параметра -v.
PYTHONUSERBASE	Корневой каталог пользовательской библиотеки пакетов.
PYTHONCASEOK	Если определена, интерпретатор будет импортировать модули без учета регистра символов в их именах.

Переменная PYTHONPATH определяет путь поиска модулей, добавляемый в начало списка sys.path, который описывается в главе 9. Переменная PYTHONSTARTUP определяет путь к файлу, который будет выполняться при запуске интерпретатора в интерактивном режиме. Переменная PYTHONHOME используется, чтобы задать каталог установки Python, но она редко бывает нужна, так как интерпретатор знает, где находятся его библиотеки и каталог site-packages, куда обычно устанавливаются расширения. Если указан единственный каталог, например /usr/local, интерпретатор будет искать все файлы в этом каталоге. Если указано два каталога, например: /usr/local:/usr/local/sparc-solaris-2.6, интерпретатор будет искать платформонезависимые файлы в первом каталоге, а платформо-зависимые – во втором. Переменная PYTHONHOME не оказывает никакого влияния, если выполняемый файл интерпретатора Python отсутствует в указанном каталоге.

Переменная окружения PYTHONIOENCODING может представлять интерес для пользователей Python 3, потому что с ее помощью можно определить ко-

дировку и политику обработки ошибок при работе со стандартными потоками ввода-вывода. В некоторых случаях это может оказаться важным, потому что при работе в интерактивном режиме Python 3 напрямую выводит текст в Юникоде. Это может, в свою очередь, вызывать неожиданные исключения при простом исследовании данных. Например:

```
>>> a = 'Jalape\xf1o'
>>> a
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/tmp/lib/python3.0/io.py", line 1486, in write
    b = encoder.encode(s)
  File "/tmp/lib/python3.0/encodings/ascii.py", line 22, in encode
    return codecs.ascii_encode(input, self.errors)[0]
UnicodeEncodeError: 'ascii' codec can't encode character '\xf1' in position 7:
ordinal not in range(128)
(Перевод:
Трассировочная информация (самый последний вызов - самый нижний):
  Файл "<stdin>", строка 1, в <module>
  Файл "/tmp/lib/python3.0/encodings/ascii.py", строка 22, в encode
    return codecs.ascii_encode(input, self.errors)[0]
UnicodeEncodeError: невозможно преобразовать символ '\xf1', в позиции 7,
в кодировку 'ascii': порядковое значение за пределами range(128)
)
>>>
```

Чтобы исправить эту проблему, можно присвоить переменной окружения PYTHONIOENCODING значение, такое как 'ascii:backslashreplace' или 'utf-8'. Результат будет следующим:

```
>>> a = 'Jalape\xf1o'
>>> a
'Jalape\xf1o'
>>>
```

В Windows значения некоторых переменных окружения, таких как PYTHONPATH, дополняются значениями из записей в реестре, находящихся в ветке HKEY_LOCAL_MACHINE/Software/Python.

Интерактивные сеансы

Если имя программы не указано и стандартный ввод интерпретатора связан с интерактивным терминалом, Python запускается в интерактивном режиме. В этом режиме выводится начальное сообщение и приглашение к вводу. Кроме того, интерпретатор выполняет сценарий, путь к которому указан в переменной окружения PYTHONSTARTUP (если эта переменная установлена). Этот сценарий выполняется, как если бы он был частью вводимой программы (то есть он не загружается с помощью инструкции import). Одно из назначений этого сценария может заключаться в том, чтобы прочитать пользовательский файл с настройками, такой как .pythonrc.

После ввода строки пользователем интерпретатор может вывести следующее приглашение к вводу одного из двух видов. Приглашение >>> появля-

ется перед вводом новой инструкции; приглашение ... показывает, что продолжается ввод предыдущей инструкции. Например:

```
>>> for i in range(0,4):
...     print i,
...
0 1 2 3
>>>
```

При желании имеется возможность изменить внешний вид приглашений, для чего достаточно изменить значения переменных sys.ps1 и sys.ps2.

В некоторых системах интерпретатор Python может быть скомпонован с библиотекой GNU readline. Эта библиотека реализует возможность сохранения истории команд, автозавершение и другие дополнения к интерактивному режиму Python.

По умолчанию вывод от команд, выполняемых в интерактивном режиме, воспроизводится с помощью функции repr(). Однако в переменной sys.displayhook можно указать другую функцию, ответственную за вывод результатов. Ниже приводится пример усечения длинных результатов:

```
>>> def my_display(x):
...     r = repr(x)
...     if len(r) > 40: print(r[:40]+"..."+r[-1])
...     else: print(r)
>>> sys.displayhook = my_display
>>> 3+4
7
>>> range(100000)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 1...]
>>>
```

Наконец, полезно знать, что при работе в интерактивном режиме результат последней операции сохраняется в специальной переменной (_). Эта переменная может использоваться для получения промежуточных результатов в последовательностях операций. Например:

```
>>> 7 + 3
10
>>> _ + 2
12
>>>
```

Установка значения переменной _ происходит в функции displayhook(), показанной выше. Поэтому если вы переопределяете функцию displayhook(), ваша функция должна устанавливать значение переменной _, если, конечно, вы желаете сохранить эту функциональность.

Запуск приложений на языке Python

В большинстве случаев желательно, чтобы программы запускали интерпретатор автоматически, а не требовалось запускать его вручную. В UNIX этого можно добиться, если присвоить файлу с программой разрешение на

выполнение и указать в первой строке программы путь к интерпретатору, например:

```
#!/usr/bin/env python
# Python code from this point on...
print “Привет, Мир!”
...
```

В Windows двойной щелчок на файле с расширением .py, .pyw, .wpy, .pyc или .pyo автоматически будет запускать интерпретатор. Обычно программы запускаются в окне консоли, если они не имеют расширение .pyw (в этом случае программы запускаются как приложения с графическим интерфейсом). Чтобы передать интерпретатору дополнительные параметры командной строки, можно запускать программу из файла .bat. Например, следующий файл .bat просто запускает сценарий на языке Python и передает интерпретатору параметры, полученные файлом .bat из командной строки:

```
:: foo.bat
:: Запускает сценарий foo.py и передает интерпретатору
:: полученные параметры командной строки
c:\python26\python.exe c:\pythonscripts\foo.py %*
```

Файлы с настройками местоположения библиотек

Обычно в состав установки Python может входить множество сторонних модулей и пакетов. Чтобы настроить доступ к этим пакетам, интерпретатор сначала импортирует модуль site. Роль модуля site заключается в том, чтобы отыскать файлы пакетов и добавить дополнительные каталоги в путь поиска модулей sys.path. Кроме того, модуль site устанавливает кодировку по умолчанию, которая используется для преобразования строк Юникода.

Модуль site сначала создает список имен каталогов, конструируя его из значений переменных sys.prefix и sys.exec_prefix, как показано ниже:

```
[ sys.prefix,               # Только для Windows
  sys.exec_prefix,          # Только для Windows
  sys.prefix + ‘lib/pythonvers/site-packages’,
  sys.prefix + ‘lib/site-python’,
  sys.exec_prefix + ‘lib/pythonvers/site-packages’,
  sys.exec_prefix + ‘lib/site-python’ ]
```

Дополнительно, если разрешено, в этот список могут добавляться каталоги пакетов, определяемые пользователем (описываются в следующем разделе).

Для каждого элемента в списке проверяется наличие указанного каталога. Если каталог существует, он добавляется в переменную sys.path. Затем проверяется наличие каких-либо файлов с настройками пути поиска (файлов с расширением .pth). Файлы с настройками пути поиска могут содержать списки путей к каталогам, zip-файлам или к файлам с расширением .egg, относительно каталога, где находится файл с настройками, которые должны быть добавлены в sys.path. Например:

```
# файл 'foo.pth' с настройками пакета foo
foo
bar
```

Каждый каталог в файле с настройками пути поиска должен находиться в отдельной строке. Комментарии и пустые строки игнорируются. Когда модуль `site` загружает файл, он проверяет существование каждого каталога. Если каталог существует, он добавляется в переменную `sys.path`. Повторяющиеся элементы добавляются только один раз.

После того как все каталоги будут добавлены в `sys.path`, предпринимается попытка импортировать модуль `sitecustomize`. Цель этого модуля состоит в том, чтобы выполнить дополнительные настройки путей поиска. Если попытка импортировать модуль `sitecustomize` закончится исключением `ImportError`, эта ошибка просто игнорируется. Импортирование модуля `sitecustomize` производится перед добавлением пользовательских каталогов в `sys.path`. Поэтому попытка поместить этот файл в каталог пользователя не даст желаемого эффекта.

Кроме того, модуль `site` отвечает за настройку кодировки по умолчанию, используемой при преобразовании строк Юникода. Изначально выбирается кодировка `'ascii'`. Однако ее можно изменить, поместив в файл `sitecustomize.py` программный код, вызывающий функцию `sys.setdefaultencoding()` с другой кодировкой, такой как `'utf-8'`. При желании можно изменить исходный текст модуля `site` так, чтобы он автоматически устанавливал кодировку, опираясь на системные настройки.

Местоположение пользовательских пакетов

Как правило, сторонние модули устанавливаются так, чтобы они были доступны всем пользователям. Однако пользователи могут создавать свои собственные каталоги для размещения модулей и пакетов. В системах UNIX и Macintosh эти каталоги обычно находятся в каталоге `~/.local` и называются примерно так: `~/.local/lib/python2.6/site-packages`. В Windows местоположение этого каталога определяется по содержимому переменной окружения `%APPDATA%`, которая обычно имеет примерно такое значение: `C:\Documents and Settings\David Beazley\Application Data`. Внутри этого каталога можно найти подкаталог `"Python\Python26\site-packages"`.

Если пользователь создает собственные модули и пакеты, которые он хотел бы использовать как библиотеки, эти модули можно поместить в каталог с библиотекой пользователя. Сторонние модули также можно установить в этот каталог, указав параметр `--user` при запуске сценария `setup.py`. Например: `python setup.py install --user`.

Включение будущих особенностей

Новые особенности языка, которые оказывают влияние на совместимость с более старыми версиями Python, в первых выпусках, где они представлены, обычно отключены. Включить эти особенности можно с помощью инструкции `from __future__ import`. Например:

```
# Включить новую семантику оператора деления
from __future__ import division
```

В случае использования эта инструкция должна быть первой инструкцией в модуле или в программе. Кроме того, область видимости особенностей, импортированных из `__future__`, ограничивается только областью видимости модуля, в котором была использована эта инструкция. Поэтому импортированные особенности не будут оказывать влияния на поведение модулей из библиотеки Python или на устаревший программный код, который опирается на прежнее поведение интерпретатора.

В табл. 10.3 перечислены особенности, которые существуют на настоящий момент:

Таблица 10.3. Имена особенностей в модуле __future__

Имя особенности	Описание
`nested_scopes`	Поддержка вложенных областей видимости в функциях. Впервые появилась в Python 2.1 и используется по умолчанию, начиная с версии Python 2.2.
`generators`	Поддержка генераторов. Впервые появилась в Python 2.2 и используется по умолчанию, начиная с версии Python 2.3.
`division`	Измененная семантика оператора деления, когда при делении целых чисел возвращается дробное число. Например, выражение `1/4` вернет `0.25` вместо `0`. Впервые появилась в Python 2.2 и оставалась отключенной вплоть до версии 2.6. Используется по умолчанию, начиная с версии Python 3.0.
`absolute_import`	Измененное поведение операции импортирования относительно пакетов. В настоящее время, когда модуль пакета использует инструкцию импорта, такую как `import string`, поиск сначала выполняется в текущем каталоге пакета, а затем в каталогах, перечисленных в переменной `sys.path`. Однако это делает невозможным использование библиотечных модулей, имена которых совпадают с именами модулей в пакете. Если включить эту особенность, инструкция `import` *`module`* будет выполнять импортирование по абсолютному пути. То есть такая инструкция, как `import string`, всегда будет загружать модуль `string` из стандартной библиотеки. Впервые появилась в Python 2.5 и оставалась отключенной вплоть до версии 2.6. Используется по умолчанию, начиная с версии Python 3.0.
`with_statement`	Поддержка менеджеров контекста и инструкции `with`. Впервые появилась в Python 2.5 и используется по умолчанию, начиная с версии Python 2.6.
`print_function`	Вместо инструкции `print` использует функцию `print()`. Впервые появилась в Python 2.6 и используется по умолчанию, начиная с версии Python 3.0.

Следует отметить, что имена особенностей никогда не удалялись из модуля `__future__`. Благодаря этому, даже если та или иная особенность включена

по умолчанию в более поздних версиях Python, это не повлечет за собой нарушение работоспособности существующего программного кода, использующего имена этих особенностей.

Завершение программы

Программа **завершается, когда в файле не остается невыполненных инструкций**, когда возбуждается исключение `SystemExit` (обычно функцией `sys.exit()`) или когда интерпретатор принимает сигнал `SIGTERM` или `SIGHUP` (в UNIX). На выходе интерпретатор уменьшает счетчики ссылок во всех объектах во всех существующих пространствах имен (и уничтожает эти пространства имен). Если счетчик ссылок объекта достигает нуля, объект уничтожается и при этом вызывается его метод `__del__()`.

Важно отметить, что в некоторых случаях в процессе завершения программы метод `__del__()` может не вызываться. Это может происходить из-за наличия циклических ссылок между объектами (в этом случае объекты продолжают существовать в памяти, но они будут недоступны после уничтожения пространства имен). Хотя сборщик мусора в интерпретаторе способен определять неиспользуемые циклические ссылки в процессе выполнения, он обычно не вызывается при завершении программы.

Поскольку нет никаких гарантий, что метод `__del__()` будет вызван при завершении программы, лучше всего предусмотреть явное освобождение некоторых ресурсов, таких как открытые файлы и сетевые соединения. Для этого в пользовательские объекты можно добавить специализированные методы (например, `close()`). Другая возможность заключается в том, чтобы написать функцию завершения и зарегистрировать ее с помощью модуля `atexit`, как показано ниже:

```
import atexit
connection = open_connection("deaddot.com")

def cleanup():
    print "Завершение..."
    close_connection(connection)

atexit.register(cleanup)
```

Подобным способом можно также вызвать сборщик мусора:

```
import atexit, gc
atexit.register(gc.collect)
```

И последнее замечание, касающееся завершения программы: методы `__del__()` некоторых объектов могут пытаться обратиться к глобальным данным или методам, объявленным в других модулях. Поскольку к этому моменту требуемые объекты уже могут быть уничтожены, в методе `__del__()` может возникать исключение `NameError`, которое приводит к появлению сообщения об ошибке, как показано ниже:

```
Exception exceptions.NameError: 'c' in <method Bar.__del__
of Bar instance at c0310> ignored
```

```
(Перевод:
Исключение exceptions.NameError: имя 'c' в <method Bar.__del__ экземпляра Bar по
адресу c0310> проигнорировано
)
```

Если происходит такая ошибка, это означает, что работа метода `__del__()` была прервана преждевременно. Это также может означать, что не была выполнена какая-то важная операция (например, не было закрыто соединение с сервером). Если подобные ошибки принципиальны, вероятно, лучше предусмотреть явное выполнение операций по завершению, а не полагаться на то, что интерпретатор уничтожит объекты начисто при выходе из программы. Исключение `NameError` можно также устранить, определив аргументы по умолчанию в объявлении метода `__del__()`:

```
import foo
class Bar(object):
    def __del__(self, foo=foo):
        foo.bar()               # Вызывается метод в модуле foo
```

В некоторых случаях бывает необходимо завершать работу программы без выполнения каких-либо заключительных действий. Добиться этого можно вызовом функции `os._exit(`*status*`)`. Эта функция представляет собой интерфейс к низкоуровневому системному вызову `exit()`, ответственному за уничтожение процесса интерпретатора Python. При вызове этой функции программа завершается немедленно, без выполнения дополнительных действий.

11

Тестирование, отладка, профилирование и оптимизация

В отличие от программ, написанных на таких языках, как C или Java, программы на языке Python не обрабатываются компилятором, который создает выполняемую программу. В этих языках программирования компилятор является первой линией обороны от программных ошибок – он отыскивает такие ошибки, как вызов функций с недопустимым количеством аргументов или присваивание недопустимых значений переменным (то есть выполняет проверку типов). В языке Python такие проверки выполняются только после запуска программы. Поэтому невозможно сказать, содержит программа ошибки или нет, пока она не будет запущена и протестирована. Но и это еще не все, – если не опробовать программу всеми возможными способами, когда поток управления пройдет все возможные ветви программы, всегда остается вероятность, что в программе скрыта какая-нибудь ошибка, которая ждет своего часа (к счастью, такие ошибки обычно обнаруживаются уже через несколько дней *после* передачи программы пользователю).

В этой главе рассматриваются приемы и библиотечные модули, используемые для тестирования, отладки и профилирования программного кода на языке Python, способные помочь в решении подобных проблем. В конце главы обсуждаются некоторые стратегии по оптимизации программного кода.

Строки документирования и модуль doctest

Если первая строка функции, класса или модуля является строкой, эта строка называется *строкой документирования*. Включение строк документирования считается хорошим тоном, потому что эти строки используются в качестве источников информации различными инструментами разработки. Например, строки документирования можно просматривать с помощью команды `help()`, а также средствами интегрированной среды разработки на языке Python. Программисты обычно просматривают содержимое строк документирования при проведении экспериментов в ин-

терактивной оболочке, поэтому в эти строки принято включать короткие примеры сеансов работы с интерактивной оболочкой. Например:

```
# splitter.py
def split(line, types=None, delimiter=None):
    """Разбивает текстовую строку и при необходимости
       выполняет преобразование типов.
       Например:

       >>> split('GOOG 100 490.50')
        ['GOOG', '100', '490.50']
       >>> split('GOOG 100 490.50',[str, int, float])
        ['GOOG', 100, 490.5]
       >>>

       По умолчанию разбиение производится по пробельным символам,
       но имеется возможность указать другой символ-разделитель, в виде
       именованного аргумента:

       >>> split('GOOG,100,490.50',delimiter=',')
        ['GOOG', '100', '490.50']
       >>>
    """
    fields = line.split(delimiter)
    if types:
        fields = [ ty(val) for ty,val in zip(types,fields) ]
    return fields
```

Типичная проблема, связанная с документацией, состоит в том, чтобы обеспечить ее соответствие актуальной реализации функции. Например, программист может изменить функцию и забыть обновить документацию.

Помощь в решении этой проблемы может оказать модуль `doctest`. Модуль `doctest` собирает строки документирования, просматривает их на наличие примеров интерактивных сеансов и выполняет эти примеры, как последовательность тестов. Чтобы воспользоваться модулем `doctest`, обычно требуется создать отдельный модуль, который будет выполнять тестирование. Например, если предположить, что реализация функции из предыдущего примера находится в файле `splitter.py`, можно было бы создать файл `testsplitter.py` со следующим содержимым:

```
# testsplitter.py
import splitter
import doctest

nfail, ntests = doctest.testmod(splitter)
```

В этом фрагменте вызов `doctest.testmod(`*`module`*`)` запускает процесс тестирования указанного модуля *`module`* и возвращает количество ошибок и общее количество выполненных тестов. Если все тесты прошли успешно, ничего не выводится. В противном случае на экране появится отчет об ошибках, в котором будут показаны различия между ожидаемыми и фактическими результатами. Если потребуется получить более подробный отчет о тестировании, можно вызвать функцию в виде `testmod(`*`module`*`, verbose=True)`.

В противоположность созданию отдельного файла, выполнение тестов можно реализовать непосредственно в библиотечных модулях, для чего достаточно включить в конец файла модуля следующий программный код:

```
...
if __name__ == '__main__':
    # тестирование самого себя
    import doctest
    doctest.testmod()
```

После этого тестирование, основанное на содержимом строк документирования, будет выполняться, если запустить модуль как самостоятельную программу. В противном случае, при импортировании файла, тесты будут игнорироваться.

При тестировании функций модуль `doctest` ожидает вывод, в точности совпадающий с тем, что будет получен в интерактивной оболочке. В результате такой вид тестирования очень чувствителен к лишним или отсутствующим пробельным символам и к точности представления чисел. В качестве примера рассмотрим следующую функцию:

```
def half(x):
    """Возвращает половину x. Например:

    >>> half(6.8)
    3.4
    >>>
    """
    return x/2
```

Если провести тестирование этой функции с помощью модуля `doctest`, будет получен следующий отчет об ошибках:

```
**********************************************************************
File "half.py", line 4, in __main__.half
Failed example:
    half(6.8)
Expected:
    3.4
Got:
    3.3999999999999999
**********************************************************************
(Перевод:
**********************************************************************
Файл "half.py", строка 4, в __main__.half
Ошибочный пример:
    half(6.8)
Ожидалось:
    3.4
Получено:
    3.3999999999999999
**********************************************************************
)
```

Чтобы исправить эту проблему, необходимо либо привести строку документирования в точное соответствие с получаемыми результатами, либо привести в документации более удачный пример.

Модуль doctest чрезвычайно прост в использовании, поэтому не может быть никаких оправданий, чтобы не использовать его в своих программах. Однако имейте в виду, что модуль doctest — не тот инструмент, который можно использовать для полного тестирования программы. Применение этого модуля для полного тестирования может привести к чрезмерному раздуванию и усложнению строк документирования, что снижает полезность документации (пользователь наверняка будет недоволен, если он обратится за справочной информацией, а в ответ ему будет представлен список из 50 примеров, охватывающих все хитрости использования функции). Для такого тестирования предпочтительнее использовать модуль unittest.

Наконец, модуль doctest имеет огромное количество параметров настройки различных аспектов, определяющих, как выполнять тестирование и как отображать результаты. Поскольку в большинстве типичных случаев использования модуля эти параметры не требуют изменения, они здесь не рассматриваются. Дополнительную информацию по этой теме можно найти по адресу: *http://docs.python.org/library/doctest.html*.

Модульное тестирование и модуль unittest

Для более полноценного тестирования программ можно использовать модуль unittest. При модульном тестировании разработчик пишет набор обособленных тестов для каждого компонента программы (например, для отдельных функций, методов, классов и модулей). Затем эти тесты используются для проверки корректности поведения основных компонентов, составляющих крупные программы. По мере роста программы в размерах модульные тесты для различных компонентов могут объединяться в крупные структуры и средства тестирования. Это может существенно упростить задачу проверки корректности поведения, а также определения и исправления проблем по мере их появления. Использование этого модуля иллюстрирует следующий фрагмент программного кода, взятый из предыдущего раздела:

```
# splitter.py
def split(line, types=None, delimiter=None):
    """Разбивает текстовую строку и при необходимости
       выполняет преобразование типов.
       ...
    """
    fields = line.split(delimiter)
    if types:
        fields = [ ty(val) for ty,val in zip(types,fields) ]
    return fields
```

Если потребуется написать модульные тесты для проверки различных аспектов применения функции split(), можно создать отдельный модуль testsplitter.py, например:

```
# testsplitter.py
import splitter
import unittest

# Модульные тесты
class TestSplitFunction(unittest.TestCase):
    def setUp(self):
        # Выполнить настройку тестов (если необходимо)
        pass
    def tearDown(self):
        # Выполнить завершающие действия (если необходимо)
        pass
    def testsimplestring(self):
        r = splitter.split('GOOG 100 490.50')
        self.assertEqual(r,['GOOG','100','490.50'])
    def testtypeconvert(self):
        r = splitter.split('GOOG 100 490.50',[str, int, float])
        self.assertEqual(r,['GOOG', 100, 490.5])
    def testdelimiter(self):
        r = splitter.split('GOOG,100,490.50',delimiter=',')
        self.assertEqual(r,['GOOG','100','490.50'])

# Запустить тестирование
if __name__ == '__main__':
    unittest.main()
```

Чтобы запустить тестирование, достаточно просто запустить интерпретатор Python, передав ему файл `testsplitter.py`. Например:

```
% python testsplitter.py
...
----------------------------------------------------------------------
Run 3 tests in 0.014s

OK
```

В своей работе модуль `unittest` опирается на объявление класса, производного от класса `unittest.TestCase`. Отдельные тесты определяются как методы, имена которых начинаются со слова `'test'`, например `'testsimplestring'`, `'testtypeconvert'` и так далее. (Важно отметить, что имена методов могут выбираться произвольно, главное, чтобы они начинались со слова `'test'`.) Внутри каждого теста выполняются проверки различных условий.

Экземпляр *t* класса `unittest.TestCase` имеет следующие методы, которые могут использоваться для тестирования и управления процессом тестирования:

t`.setUp()`

Вызывается для выполнения настроек, перед вызовом любых методов тестирования.

t`.tearDown()`

Вызывается для выполнения заключительных действий после выполнения всех тестов.

t.assert_(*expr* [, *msg*])
t.failUnless(*expr* [, *msg*])

Сообщает об ошибке тестирования, если выражение *expr* оценивается как *False*. *msg* – это строка сообщения, объясняющая причины ошибки (если задана).

t.assertEqual(*x*, *y* [,*msg*])
t.failUnlessEqual(*x*, *y* [, *msg*])

Сообщает об ошибке тестирования, если *x* и *y* не равны. *msg* – это строка сообщения, объясняющая причины ошибки (если задана).

t.assertNotEqual(*x*, *y* [, *msg*])
t.failIfEqual(*x*, *y*, [, *msg*])

Сообщает об ошибке тестирования, если *x* и *y* равны. *msg* – это строка сообщения, объясняющая причины ошибки (если задана).

t.assertAlmostEqual(*x*, *y* [, *places* [, *msg*]])
t.failUnlessAlmostEqual(*x*, *y*, [, *places* [, *msg*]])

Сообщает об ошибке тестирования, если числа *x* и *y* не совпадают с точностью до знака *places* после десятичной точки. Проверка выполняется за счет вычисления разности между *x* и *y* и округления результата до указанного числа знаков *places* после десятичной точки. Если результат равен нулю, числа *x* и *y* можно считать почти равными. *msg* – это строка сообщения, объясняющая причины ошибки (если задана).

t.assertNotAlmostEqual(*x*, *y*, [, *places* [, *msg*]])
t.failIfAlmostEqual(*x*, *y* [, *places* [, *msg*]])

Сообщает об ошибке тестирования, если числа *x* и *y* совпадают с точностью до знака *places* после десятичной точки. *msg* – это строка сообщения, объясняющая причины ошибки (если задана).

t.assertRaises(*exc*, *callable*, ...)
t.failUnlessRaises(*exc*, *callable*, ...)

Сообщает об ошибке тестирования, если вызываемый объект *callable* не возбуждает исключение *exc*. Остальные аргументы методов передаются вызываемому объекту *callable*, как аргументы. Для тестирования набора исключений в аргументе *exc* передается кортеж с этими исключениями.

t.failIf(*expr* [, *msg*])

Сообщает об ошибке тестирования, если выражение *expr* оценивается как True. *msg* – это строка сообщения, объясняющая причины ошибки (если задана).

t.fail([*msg*])

Сообщает об ошибке тестирования. *msg* – это строка сообщения, объясняющая причины ошибки (если задана).

t.failureException

В этом атрибуте сохраняется последнее исключение, перехваченное в тесте. Может использоваться, когда необходимо не только проверить, что ис-

ключение возбуждается, но что при этом оно сопровождается требуемым значением, – например, когда необходимо проверить сообщение, генерируемое исключением.

Следует отметить, что модуль `unittest` имеет огромное количество дополнительных параметров настройки, используемых для группировки тестов, создания наборов тестов и управления окружением, в котором выполняются тесты. Эти особенности не имеют прямого отношения к процессу создания тестов (классы обычно пишутся независимо от того, как в действительности выполняются тесты). В документации, по адресу *http://docs.python.org/library/unittest.html*, можно найти дополнительную информацию о том, как организовать тесты для крупных программ.

Отладчик Python и модуль pdb

В состав Python **входит простой отладчик командной строки, который реа**лизован в виде модуля `pdb`. Модуль `pdb` поддерживает возможность постварийной отладки, исследования кадров стека, установки точек останова, выполнения исходного программного кода в пошаговом режиме и вычисления выражений.

Существует несколько функций для вызова отладчика из программы или из интерактивной оболочки Python.

`run(`*`statement`* `[, `*`globals`* `[, `*`locals`*`]])`

Выполняет строку *`statement`* под управлением отладчика. Приглашение к вводу отладчика появляется непосредственно перед выполнением какого-либо программного кода. Ввод команды `'continue'` инициирует выполнение этого кода. Аргументы *`globals`* и *`locals`* определяют глобальное и локальное пространство имен соответственно, в котором будет выполняться программный код.

`runeval(`*`expression`* `[, `*`globals`* `[, `*`locals`*`]])`

Вычисляет выражение в строке *`expression`* под управлением отладчика. Приглашение к вводу отладчика появляется непосредственно перед выполнением какого-либо программного кода, поэтому чтобы вычислить значение выражения, необходимо ввести команду `'continue'`, которая запустит функцию `run()`. В случае успеха возвращается значение выражения.

`runcall(`*`function`* `[, `*`argument`*`, ...])`

Вызовет функцию *`function`* под управлением отладчика. Аргумент *`function`* должен быть вызываемым объектом. Дополнительные аргументы передаются функции *`function`*. Приглашение к вводу отладчика появляется непосредственно перед выполнением какого-либо программного кода. По завершении возвращается значение функции *`function`*.

`set_trace()`

Запускает отладчик в точке вызова этой функции. Может использоваться для создания точек останова в интересующих местах программы.

`post_mortem(traceback)`

Запускает поставарийную отладку с использованием объекта *traceback*, содержащего трассировочную информацию. Объект *traceback* обычно можно получить с помощью такой функции, как `sys.exc_info()`.

`pm()`

Переходит в режим поставарийной отладки с использованием объекта *traceback*, сгенерированного последним исключением.

Из всех функций, вызывающих отладчик, самой простой в использовании, пожалуй, является функция `set_trace()`. Когда при работе со сложным приложением выявляется какая-либо проблема, можно просто вставить вызов `set_trace()` в программный код и запустить приложение. Когда интерпретатор встретит этот вызов, выполнение программы будет приостановлено и управление будет передано отладчику, с помощью которого можно будет исследовать окружение, в котором протекает выполнение программы. Выполнение программы будет продолжено сразу же после выхода из отладчика.

Команды отладчика

После запуска отладчика выводится приглашение к вводу `(Pdb)`, как показано ниже:

```
>>> import pdb
>>> import buggymodule
>>> pdb.run('buggymodule.start()')
> <string>(0)?()
(Pdb)
```

`(Pdb)` – это приглашение к вводу отладчика, в котором распознаются следующие команды. Обратите внимание, что некоторые команды имеют две формы – краткую и длинную. Для обозначения обеих форм в описании команд использованы круглые скобки. Например, `h(elp)` означает, что можно ввести команду `h` или `help`.

`[!]statement`

Выполняет инструкцию *statement* (однострочную) в контексте текущего кадра стека. Символ восклицательного знака можно опустить, но его использование обязательно, если первое слово инструкции *statement* совпадает с командой отладчика. Чтобы определить глобальную переменную, можно предварить инструкцию присваивания командой «`global`» в той же строке:

```
(Pdb) global list_options; list_options = ['-l']
(Pdb)
```

`a(rgs)`

Выводит список аргументов текущей функции.

`alias [name [command]]`

Создает псевдоним *name* для команды *command*. Подстроки, такие как `'%1','%2'` и так далее, в строке *command* замещаются значениями параметров, которые

указываются при вводе псевдонима. Подстрока '%*' соответствует всему списку параметров. Если значение *command* не задано, выводится текущее определение **псевдонима. Псевдонимы допускается вкладывать друг в друга** и они могут содержать все, что допускается вводить в приглашении Pdb. Например:

```
# Выводит переменные экземпляров (порядок использования: "pi classInst")
alias pi for k in %1._ _dict_ _.keys(): print "%1.",k,"=",%1._ _dict_ _[k]
# Выводит переменные для экземпляра self
alias ps pi self
```

b(reak) [*loc* [, *condition*]]

Устанавливает точку останова в местоположении *loc*. Значением *loc* может быть либо имя файла и номер строки, либо имя функции в модуле. Синтаксис параметра имеет следующий вид:

Значение	Описание
n	Номер строки в текущем файле
filename:n	Номер строки в другом файле
function	Имя функции в текущем модуле
module.function	Имя функции в другом модуле

Если параметр *loc* не задан, выводится список всех точек останова, установленных на текущий момент. *condition* – это выражение, которое должно оцениваться как истина, чтобы произошел останов в данной точке. Каждой точке останова присваивается свой номер, который выводится по завершении этой команды. Эти номера можно использовать в некоторых других командах отладчика, описываемых ниже.

cl(ear) [*bpnumber* [*bpnumber* ...]]

Сбрасывает точки останова с указанными номерами *bpnumber*. Если номера не указываются, команда сбросит все точки останова.

commands [*bpnumber*]

Определяет последовательность команд отладчика для автоматического выполнения по достижении точки останова *bpnumber*. Если необходимо указать несколько команд, их можно ввести в нескольких строках и использовать слово end, как признак конца последовательности. Если включить в последовательность команду continue, после встречи точки останова выполнение программы будет продолжено автоматически. Если параметр *bpnumber* не задан, команда commands применяется к последней установленной точке останова.

condition *bpnumber* [*condition*]

Добавляет условие *condition* к точке останова *bpnumber*. Параметр *condition* – это выражение, значение которого должно оцениваться как истинное, чтобы произошел останов в данной точке. Отсутствие параметра *condition* приводит к сбросу всех условий, установленных ранее.

`c(ont(inue))`

Возобновляет выполнение программы, пока не будет встречена следующая точка останова.

`disable [`*`bpnumber`* `[`*`bpnumber`* `...]]`

Деактивирует указанные точки останова. В отличие от команды `clear`, после команды `disable` имеется возможность вновь активировать эти точки останова.

`d(own)`

Перемещает текущий кадр стека на один уровень вниз в стеке трассировки.

`enable [`*`bpnumber`* `[`*`bpnumber`* `...]]`

Активирует указанные точки останова.

`h(elp) [`*`command`*`]`

Выводит список доступных команд. Если указана команда *`command`*, возвращает справочную информацию по этой команде.

`ignore` *`bpnumber`* `[`*`count`*`]`

Деактивирует точку останова на *`count`* проходов.

`j(ump)` *`lineno`*

Выполняет переход к следующей строке. Может использоваться только для перехода между инструкциями в одном кадре стека. Кроме того, не позволяет выполнить переход внутрь некоторых инструкций, например в середину цикла.

`l(ist) [`*`first`* `[, `*`last`*`]]`

Выводит листинг исходного программного кода. При использовании без аргументов эта команда выведет 11 строк, окружающих текущую строку (5 строк до и 5 строк после). При использовании с единственным аргументом она выведет 11 строк, начиная с указанной строки. При использовании с двумя аргументами – выведет строки из указанного диапазона. Если значение параметра *`last`* меньше значения параметра *`first`*, оно будет интерпретироваться, как счетчик строк.

`n(ext)`

Выполняет инструкции до следующей строки в текущей функции. Если в текущей строке присутствуют вызовы других функций, они не учитываются.

`p` *`expression`*

Вычисляет значение выражения *`expression`* в текущем контексте и выводит его.

`pp` *`expression`*

То же, что и команда `p`, но результат форматируется с использованием модуля `pprint`.

`q(uit)`

Выход из отладчика.

`r(eturn)`

Выполняет инструкции до момента выхода из текущей функции.

`run [`*`args`*`]`

Перезапускает программу с аргументами командной строки *args*, которые записываются в переменную `sys.argv`. Все точки останова и другие настройки отладчика сохраняются.

`s(tep)`

Выполняет одну строку исходного программного кода и останавливает выполнение внутри вызываемых функций.

`tbreak [`*`loc`*` [, `*`condition`*`]]`

Устанавливает временную точку останова, которая удаляется после первого срабатывания.

`u(p)`

Перемещает текущий кадр стека на один уровень вверх в стеке трассировки.

`unalias `*`name`*

Удаляет указанный псевдоним.

`until`

Продолжает выполнение программы, пока поток выполнения не покинет текущий кадр стека или пока не будет достигнута строка с номером, больше чем у текущей. Например, если останов произошел в последней строке тела цикла, команда `until` продолжит выполнение всех инструкций, составляющих цикл, пока он не завершится.

`w(here)`

Выведет трассировку стека.

Отладка из командной строки

Альтернативный метод отладки заключается в том, чтобы вызвать отладчик из командной строки. Например:

```
% python -m pdb someprogram.py
```

В данном случае отладчик будет запущен автоматически непосредственно перед запуском программы, что позволит установить точки останова и внести какие-либо изменения в настройки. Чтобы запустить программу, достаточно просто ввести команду `continue`. Например, если потребуется отладить функцию `split()` в программе, где она используется, это можно сделать, как показано ниже:

```
% python -m pdb someprogram.py
> /Users/beazley/Code/someprogram.py(1)<module>()
```

```
-> import splitter
(Pdb) b splitter.split
Breakpoint 1 at /Users/beazley/Code/splitter.py:1
(Pdb) c
> /Users/beazley/Code/splitter.py(18)split()
-> fields = line.split(delimiter)
(Pdb)
```

Настройка отладчика

Если в текущем каталоге или в домашнем каталоге пользователя присутствует файл `.pdbrc`, его содержимое будет прочитано и выполнено, как если бы строки из этого файла вводились в приглашении к вводу отладчика. Это может быть использовано для того, чтобы задать команды отладчика, которые желательно выполнять каждый раз, когда запускается отладчик (чтобы каждый раз не вводить эти команды вручную).

Профилирование программы

Для сбора профилирующей информации используются модули `profile` и `cProfile`. Оба модуля действуют совершенно одинаково, разница лишь в том, что модуль `cProfile` реализован как расширение на языке C, он работает значительно быстрее и более современный. Оба модуля могут использоваться как для сбора общей информации (позволяющей выяснить, какие функции вызывались), так и для сбора статистических данных о производительности. Самый простой способ профилирования программы заключается в том, чтобы запустить ее из командной строки, как показано ниже:

```
% python -m cProfile someprogram.py
```

Как вариант, можно использовать функцию из модуля `profile`:

```
run(command [, filename])
```

Она выполняет содержимое строки *`command`* с помощью инструкции `exec` под управлением профилировщика. Аргумент *`filename`* – это имя файла для сохранения первичных данных профилирования. Если этот аргумент отсутствует, отчет выводится в поток стандартного вывода.

Результатом работы профилировщика является отчет, такой как приведенный ниже:

```
126 function calls (6 primitive calls) in 5.130 CPU seconds
Ordered by: standard name
ncalls  tottime  percall  cumtime  percall  filename:lineno(function)
     1    0.030    0.030    5.070    5.070  <string>:1(?)
 121/1    5.020    0.041    5.020    5.020  book.py:11(process)
     1    0.020    0.020    5.040    5.040  book.py:5(?)
     2    0.000    0.000    0.000    0.000  exceptions.py:101(__init__)
     1    0.060    0.060    5.130    5.130  profile:0(execfile('book.py'))
     0    0.000             0.000           profile:0(profiler)
```

Столбцы в отчете, сгенерированном функцией `run()`, интерпретируются следующим образом:

Значение	Описание
`primitive calls`	Количество нерекурсивных вызовов функций
`ncalls`	Общее число вызовов (включая рекурсивные)
`tottime`	Время, потраченное на выполнение этих функций (не включая вложенные вызовы функций)
`percall`	`tottime / ncalls`
`cumtime`	Общее время, потраченное на выполнение функций
`percall`	`cumtime / primitive calls`
`filename``:`*`lineno`*`(`*`function`*`)`	Местонахождение и имя каждой функции

Когда в первом столбце выводится два числа (например, `"121/1"`), второе из них – это число простых вызовов (primitive calls), а первое – фактическое число вызовов.

В большинстве случаев для применения этого модуля бывает вполне достаточно простого знакомства с отчетом, сгенерированным профилировщиком, например, когда требуется всего лишь определить, как распределяется время выполнения программы. Однако если потребуется сохранить данные и затем проанализировать их, можно воспользоваться модулем `pstats`. Более подробную информацию о том, как сохранять и анализировать данные профилировщика, можно найти на странице *http://docs.python.org/library/profile.html*.

Настройка и оптимизация

В этом разделе рассматриваются некоторые основные правила, соблюдение которых позволит создавать программы, работающие быстрее и потребляющие меньше памяти. Приемы, описываемые здесь, ни в коем случае не являются исчерпывающими, но они позволят программистам получить некоторую оценку своего программного кода.

Измерение производительности

Если вам потребуется узнать, как долго выполняется программа на языке Python, самый простой способ определить это состоит в том, чтобы запустить ее под управлением какой-нибудь утилиты, напоминающей команду `time` в UNIX. Если потребуется определить продолжительность работы блока инструкций, можно вставить вызовы функции `time.clock()`, которая возвращает текущее процессорное время, или `time.time()`, возвращающую текущее системное время. Например:

```
start_cpu = time.clock()
start_real= time.time()
инструкции
```

```
    инструкции
    end_cpu = time.clock()
    end_real = time.time()
    print("Действительное время в сек. %f" % (end_real - start_real))
    print("Процессорное время в сек. %f" % (end_cpu - start_cpu))
```

Имейте в виду, что этот прием имеет смысл применять, только если фрагмент программного кода, на котором производятся измерения, выполняется достаточно продолжительное время. Для измерения производительности отдельных инструкций можно использовать функцию `timeit(code [, setup])` из модуля `timeit`. Например:

```
>>> from timeit import timeit
>>> timeit('math.sqrt(2.0)','import math')
0.20388007164001465
>>> timeit('sqrt(2.0)','from math import sqrt')
0.14494490623474121
```

В этом примере первым аргументом функции `timeit()` передается программный код, продолжительность работы которого требуется измерить. Второй аргумент – это инструкция, которая будет выполнена один раз для настройки окружения. Функция `timeit()` выполняет указанную инструкцию один миллион раз и выводит время, потребовавшееся на ее выполнение. Количество повторений можно изменять с помощью именованного аргумента `number=count` функции `timeit()`.

Кроме того, в модуле `timeit` имеется функция `repeat()`, которая также может использоваться для измерения производительности. Эта функция действует точно так же, как функция `timeit()`, за исключением того, что она выполняет три серии измерений и возвращает список результатов. Например:

```
>>> from timeit import repeat
>>> repeat('math.sqrt(2.0)','import math')
[0.20306601524353027, 0.19715800285339355, 0.20907392501831055]
>>>
```

При измерении производительности обычно принято определять *прирост скорости*, который вычисляется, как частное от деления первоначального времени выполнения на новое время выполнения. Например, в предыдущем эксперименте, где приводятся результаты измерения производительности инструкций `sqrt(2.0)` и `math.sqrt(2.0)`, прирост скорости составил 0,20388/0,14494 или примерно 1,41 раза. Иногда это число выражают в процентах, говоря, что прирост скорости составил примерно 41 %.

Измерение объема потребляемой памяти

В модуле `sys` имеется функция `getsizeof()`, которую можно использовать для определения объема памяти (в байтах), потребляемого отдельными объектами. Например:

```
>>> import sys
>>> sys.getsizeof(1)
14
```

```
>>> sys.getsizeof("Hello World")
52
>>> sys.getsizeof([1,2,3,4])
52
>>> sum(sys.getsizeof(x) for x in [1,2,3,4])
56
```

Для контейнерных объектов, таких как списки, кортежи и словари, возвращается размер памяти, занимаемой самим контейнером, а не суммарный объем памяти, занимаемой всеми элементами контейнера. Например, в предыдущем примере для списка `[1,2,3,4]` сообщается объем памяти, который в действительности меньше, чем требуется для хранения четырех целых чисел (на каждое число требуется 14 байт памяти). Это объясняется тем, что содержимое списка не участвует в подсчетах. Для определения общего размера можно использовать функцию `sum()`, как показано здесь же.

Помните, что функция `getsizeof()` может дать лишь общее представление об использовании памяти различными объектами. В действительности интерпретатор стремится многократно использовать одни и те же объекты, опираясь на механизм подсчета ссылок, поэтому фактический объем памяти, занимаемый объектом, может оказаться намного меньше, чем можно было бы представить. Кроме того, так как расширения Python, написанные на языке C, могут распределять память за пределами интерпретатора, может оказаться сложным точно определить общий объем используемой памяти. Поэтому другим приемом определения фактического объема занимаемой памяти является исследование параметров выполняющегося процесса с помощью обозревателя системных процессов или диспетчера задач.

Откровенно говоря, лучший способ получить представление об объеме занимаемой памяти состоит в том, чтобы сесть и проанализировать программу. Если известно, что программа выделяет память для различных структур данных, а также известно, что это за структуры и какие данные будут храниться в них (целые числа, числа с плавающей точкой, строки и так далее), можно воспользоваться функцией `getsizeof()`, чтобы получить исходные данные для вычисления максимального объема памяти, необходимой для работы программы, или, по крайней мере, получить достаточно информации для выполнения ориентировочноой оценки.

Дизассемблирование

Для дизассемблирования функций, методов и классов на языке Python в низкоуровневые инструкции интерпретатора можно использовать модуль `dis`. В этом модуле имеется функция `dis()`, которая может использоваться, как показано ниже:

```
>>> from dis import dis
>>> dis(split)
2             0 LOAD_FAST           0 (line)
              3 LOAD_ATTR           0 (split)
              6 LOAD_FAST           1 (delimiter)
              9 CALL_FUNCTION       1
             12 STORE_FAST          2 (fields)
```

```
3           15 LOAD_GLOBAL              1 (types)
            18 JUMP_IF_FALSE           58 (to 79)
            21 POP_TOP

4           22 BUILD_LIST               0
            25 DUP_TOP
            26 STORE_FAST               3 (_[1])
            29 LOAD_GLOBAL              2 (zip)
            32 LOAD_GLOBAL              1 (types)
            35 LOAD_FAST                2 (fields)
            38 CALL_FUNCTION            2
            41 GET_ITER
      >>    42 FOR_ITER                25 (to 70)
            45 UNPACK_SEQUENCE          2
            48 STORE_FAST               4 (ty)
            51 STORE_FAST               5 (val)
            54 LOAD_FAST                3 (_[1])
            57 LOAD_FAST                4 (ty)
            60 LOAD_FAST                5 (val)
            63 CALL_FUNCTION            1
            66 LIST_APPEND
            67 JUMP_ABSOLUTE           42
      >>    70 DELETE_FAST              3 (_[1])
            73 STORE_FAST               2 (fields)
            76 JUMP_FORWARD             1 (to 80)
      >>    79 POP_TOP

5     >>    80 LOAD_FAST                2 (fields)
            83 RETURN_VALUE
>>>
```

Опытные программисты могут использовать эту информацию двумя способами. Во-первых, дизассемблированный листинг точно показывает, какие операции выполняются при работе функции. При внимательном изучении можно даже определить возможные способы оптимизации производительности. Во-вторых, для тех, кто пишет многопоточные программы, будет полезно ознакомиться с таким листингом, так как в нем представлены отдельные инструкции интерпретатора, каждая из которых выполняется атомарно. То есть эта информация может быть полезна при исследовании сложных проблем, связанных с состоянием «гонки за ресурсами».

Стратегии оптимизации

В следующих разделах в общих чертах описываются некоторые стратегии оптимизации, которые, по опыту автора, хорошо зарекомендовали себя при разработке программ на языке Python.

Изучите свою программу

Прежде чем приступать к оптимизации чего бы то ни было, необходимо понять, какой вклад в общий прирост скорости даст оптимизация той или иной части программы. Например, если в 10 раз повысить скорость работы функции, время работы которой составляет 10 % от общего времени работы программы, общий прирост скорости составит примерно 9–10 %. В за-

висимости от усилий, которые придется приложить, может оказаться, что такая оптимизация не стоит затраченных сил.

На первом этапе всегда желательно выполнить профилирование программного кода, который предстоит оптимизировать. Основное внимание следует уделить функциям и методам, на выполнение которых тратится большая часть времени, а не сомнительным операциям, которые вызываются лишь время от времени.

Изучите алгоритмы

Даже неоптимально реализованный алгоритм **O(n log n) выиграет соревнование** у высокооптимизированного алгоритма $O(n^3)$. Нет смысла стараться оптимизировать неэффективные алгоритмы – лучше попробовать найти более удачный алгоритм.

Используйте встроенные типы данных

Встроенные типы данных, такие как кортежи, списки и словари, целиком реализованы на языке C и являются наиболее высокооптимизированными структурами данных в языке Python. Следует активно использовать эти типы для хранения и манипулирования данными в программе и стараться избегать создавать собственные структуры, имитирующие их поведение (например, двоичные деревья, связанные списки и так далее).

При этом не следует забывать о типах в стандартной библиотеке. Некоторые библиотечные модули объявляют новые типы данных, которые в определенных случаях выигрывают по производительности у встроенных типов. Например, тип `collection.deque` по своей функциональности напоминает списки, но он обладает более оптимизированной реализацией операции вставки новых элементов в оба конца. В отличие от него, список обеспечивает высокую эффективность только при добавлении элементов в конец. Когда новый элемент вставляется в начало списка, все остальные элементы сдвигаются, чтобы освободить место. Время, затрачиваемое на сдвиг элементов, тем больше, чем больше список. Только чтобы дать вам почувствовать разницу, ниже приводится время, затраченное на выполнение операции вставки одного миллиона элементов в начало списка и в начало объекта типа `deque`:

```
>>> from timeit import timeit
>>> timeit('s.appendleft(37)',
...        'import collections; s = collections.deque()',
...        number=1000000)
0.24434304237365723
>>> timeit('s.insert(0,37)', 's = []', number=1000000)
612.95199513435364
```

Не добавляйте лишние уровни абстракции

Всякий раз, когда добавляется новый уровень абстракции или дополнительные удобства к функции или к объекту, это замедляет скорость работы программы. Следует стремиться сохранить баланс между удобством и производительностью. Например, целью добавления нового уровня абстракции зачастую является упрощение процесса программирования, что тоже важно.

В качестве простого примера рассмотрим программу, которая использует функцию `dict()` для создания словарей со строковыми ключами:

```
s = dict(name='GOOG',shares=100,price=490.10)
# s = {'name':'GOOG', 'shares':100, 'price':490.10 }
```

Этот прием может использоваться программистом, чтобы уменьшить объем ввода с клавиатуры (в этом случае не придется вводить кавычки вокруг имен ключей). Однако этот альтернативный способ создания словаря оказывается более медленным, потому что в нем присутствует дополнительный вызов функции.

```
>>> timeit("s = {'name':'GOOG','shares':100,'price':490.10}")
0.38917303085327148
>>> timeit("s = dict(name='GOOG',shares=100,price=490.10)")
0.94420003890991211
```

Если в процессе своей работы ваша программа создает миллионы словарей, то вы должны знать, что первый вариант выполняется быстрее. За редким исключением, любая особенность, которая расширяет или изменяет принцип действия существующих объектов Python, будет более медленной.

Помните, что классы и экземпляры основаны на применении словарей

Экземпляры и классы, определяемые пользователем, основаны на применении словарей. По этой причине операции поиска, изменения или удаления данных в экземпляре практически всегда выполняются медленнее, чем непосредственные операции со словарями. Если необходимо всего лишь создать структуру для хранения данных, применение словаря может оказаться более эффективным, чем создание класса.

Только чтобы продемонстрировать разницу, ниже приводится пример простого класса, используемого для хранения информации о товарах на складе:

```
class Stock(object):
    def __init__(self,name,shares,price):
        self.name = name
        self.shares = shares
        self.price = price
```

Если сравнить производительность этого класса с производительностью словаря, можно получить весьма интересные результаты. Для **начала сравним** производительность операции создания экземпляров:

```
>>> from timeit import timeit
>>> timeit("s = Stock('GOOG',100,490.10)","from stock import Stock")
1.3166780471801758
>>> timeit("s = {'name' : 'GOOG', 'shares' : 100, 'price' : 490.10 }")
0.37812089920043945
>>>
```

Разница в скорости создания новых объектов составила примерно 3,5 раза. Далее рассмотрим скорость выполнения простых вычислений:

```
>>> timeit("s.shares*s.price",
...     "from stock import Stock; s = Stock('GOOG',100,490.10)")
0.29100513458251953
>>> timeit("s['shares']*s['price']",
...     "s = {'name' : 'GOOG', 'shares' : 100, 'price' : 490.10 }")
0.23622798919677734
>>>
```

Здесь разница в скорости составила примерно 1,2 раза. Суть здесь в том, что хотя вы можете объявлять новые объекты, используя классы, это не является единственным способом работы с данными. Кортежи и словари нередко являются достаточно удачным выбором. Их использование позволит повысить производительность программы и уменьшить объем потребляемой памяти.

Используйте атрибут __slots__

Если программа создает большое количество экземпляров пользовательских классов, имеет смысл подумать об использовании атрибута `__slots__` в определении класса. Например:

```
class Stock(object):
    __slots__ = ['name','shares','price']
    def __init__(self,name,shares,price):
        self.name = name
        self.shares = shares
        self.price = price
```

Иногда атрибут `__slots__` рассматривается как средство обеспечения безопасности, потому что он ограничивает множество доступных имен атрибутов. Однако в действительности это, скорее, средство оптимизации производительности. Классы, в которых объявляется атрибут `__slots__`, не используют словарь для хранения данных экземпляра (для этих целей используется более эффективная внутренняя структура данных). Поэтому экземпляры таких классов не только используют меньше памяти, но и обладают более эффективным способом доступа к данным. В некоторых случаях простое добавление атрибута `__slots__`, без внесения каких-либо других изменений, может обеспечить заметный прирост скорости.

Однако по поводу использования атрибута `__slots__` следует сделать одно предупреждение. Добавление его в определение класса может вызвать необъяснимые нарушения в работе другого программного кода. Например, хорошо известно, что экземпляры хранят свои данные в словарях, доступных в виде атрибута `__dict__`. Когда в объявление класса добавляется атрибут `__slots__`, атрибут `__dict__` становится недоступен, что вызывает нарушения в работе программного кода, использующего его.

Избегайте использования оператора (.)

Всякий раз при попытке обращения к атрибуту объекта с помощью оператора (.) выполняется поиск требуемого имени атрибута. Например, при обращении к *x.name* сначала выполняется поиск переменной с именем "x" в текущем окружении, а затем в объекте *x* производится поиск имени

“name”. В случае пользовательских объектов поиск имени атрибута может выполняться в словаре экземпляра, в словаре класса и в словарях базовых классов.

Для вычислений, тесно связанных с использованием методов или функций в других модулях, практически всегда лучше устранить этап поиска атрибутов, предварительно поместив требуемую операцию в локальную переменную. Например, если используется операция вычисления квадратного корня, она будет выполняться быстрее, если импортирование будет выполнено как ‘from math import sqrt’, а затем будет вызываться функция ‘sqrt(x)’, чем в случае, когда функция вызывается как ‘math.sqrt(x)’. В первой части этого раздела было показано, что такой подход обеспечивает прирост скорости примерно в 1.4 раза.

Очевидно, что нет необходимости пытаться полностью избавиться от оператора (.), так как это может существенно усложнить чтение программного кода. Однако для участков программы, где производительность имеет особенно важное значение, этот прием может оказаться полезным.

Используйте исключения для обработки нетипичных случаев

Чтобы избежать ошибок, вы можете постараться добавить в программу дополнительные проверки. Например:

```
def parse_header(line):
    fields = line.split(“:”)
    if len(fields) != 2:
        raise RuntimeError(“Ошибка в заголовке”)
    header, value = fields
    return header.lower(), value.strip()
```

Однако существует более простой способ обработки ошибок, который состоит в том, чтобы позволить программе самой генерировать исключения и обрабатывать их. Например:

```
def parse_header(line):
    fields = line.split(“:”)
    try:
        header, value = fields
        return header.lower(), value.strip()
    except ValueError:
        raise RuntimeError(“Ошибка в заголовке”)
```

Если проверить производительность обеих версий на корректно сформированных заголовках, вторая версия будет выполняться примерно на 10 процентов быстрее. Блок try в случае, когда программный код не возбуждает исключение, выполняется быстрее, чем инструкция if.

Не используйте исключения для обработки типичных случаев

Не следует использовать исключения для обработки типичных случаев. Например, допустим, что имеется программа, которая интенсивно рабо-

тает со словарем, но в большинстве случаев искомые ключи отсутствуют в словаре. Теперь рассмотрим два подхода к организации работы со словарем:

```
# Подход 1 : Выполнение поиска и обработка исключения
try:
    value = items[key]
except KeyError:
    value = None

# Подход 2: Проверка наличия ключа и выполнение поиска
if key in items:
    value = items[key]
else:
    value = None
```

Простое измерение производительности показывает, что в случае обращения к несуществующему ключу второй подход выполняется быстрее более чем в 17 раз! Кроме того, второй подход выполняется почти в два раза быстрее, чем тот же алгоритм, но с использованием метода `items.get(key)`, а обусловлено это тем, что оператор `in` выполняется быстрее, чем вызов метода.

Применяйте приемы функционального программирования и итерации

Генераторы списков, выражения-генераторы, функции-генераторы, сопрограммы и замыкания оказываются намного эффективнее, чем думает большинство программистов. Обработка данных с применением генераторов списков и выражений-генераторов выполняется существенно быстрее, чем программный код, выполняющий итерации по данным вручную и производящий те же самые вычисления. Кроме того, эти операции выполняются намного быстрее, чем программный код, использующий такие функции, как `map()` и `filter()`. Генераторы позволяют не только повысить производительность, но и более эффективно использовать память.

Используйте декораторы и метаклассы

Декораторы и метаклассы обеспечивают возможность изменять поведение функций и классов. При этом, так как они выполняются во время объявления функции или класса, они могут дать немалый прирост производительности, особенно если в программе имеется множество особенностей, которые могут включаться и выключаться. В главе 6 «Функции и функциональное программирование» приводится пример использования декоратора для подключения к функциям возможности журналирования событий, который не оказывает отрицательного влияния на производительность, когда журналирование отключено.

II

Стандартная библиотека Python

12

Встроенные функции

В этой главе описываются встроенные функции и исключения языка Python. Большая часть этого материала менее формально была рассмотрена в предыдущих главах. Эта глава просто объединяет все приводившиеся ранее сведения в один раздел и дополняет их описанием более тонких особенностей некоторых функций. Кроме того, Python 2 включает в себя множество встроенных функций, которые были объявлены устаревшими и исключены из Python 3. Здесь эти функции не описываются; вместо этого основное внимание будет уделено более современным возможностям.

Встроенные функции и типы

Некоторые типы данных, функции и переменные всегда доступны интерпретатору и могут использоваться внутри любого модуля. Чтобы получить доступ к этим функциям, не требуется импортировать дополнительные модули, тем не менее все они содержатся внутри модуля `__builtin__`, в Python 2, и внутри модуля `builtins`, в Python 3. Внутри других модулей, которые могут импортироваться программой, присутствует переменная `__builtins__`, которая также ссылается на этот модуль.

`abs(`*x*`)`

Возвращает абсолютное значение *x*.

`all(`*s*`)`

Возвращает `True`, если все значения в итерируемом объекте *s* оцениваются как `True`.

`any(`*s*`)`

Возвращает `True`, если хотя бы одно значение в итерируемом объекте *s* оценивается как `True`.

`ascii(`*x*`)`

Создает печатное представление объекта *x*, точно так же, как и функция `repr()`, но в результате используются только символы ASCII. Символы, от-

сутствующие в наборе ASCII, преобразуются в соответствующие экранированные последовательности. Эта функция может использоваться для просмотра содержимого строк Юникода в терминалах или в командных оболочках, не поддерживающих Юникод. Только в Python 3.

`basestring`

Абстрактный тип данных, который является суперклассом для всех строковых типов в Python 2 (`str` и `unicode`). Используется только при тестировании на принадлежность к строковым типам. Например, `isinstance(s,basestring)` вернет `True`, если объект `s` является строкой того или иного типа. Только в Python 2.

`bin(`*x*`)`

Возвращает строку с двоичным представлением целого числа *x*.

`bool([`*x*`])`

Тип, используемый для представления логических значений `True` и `False`. Если используется для преобразования *x*, возвращает `True`, если значение объект *x* оценивается как истина, при использовании обычной семантики определения истинности (то есть ненулевое число, непустой список и так далее). В противном случае возвращает `False`. Кроме того, если функция `bool()` вызывается без аргументов, она по умолчанию возвращает `False`. Класс `bool` является производным от класса `int`, поэтому логические значения `True` и `False` можно использовать в математических операциях, как целые числа `1` и `0`.

`bytearray([`*x*`])`

Тип, используемый для представления изменяемых массивов байтов. Когда создается новый массив, аргумент *x* может быть итерируемой последовательностью целых чисел в диапазоне от 0 до 255, строкой 8-битных символов, литералом типа `bytes` или целым числом, определяющим размер массива байтов (в этом случае все элементы массива инициализируются значением `0`). Объект типа `bytearray` выглядит, как массив целых чисел. При обращении к элементу массива, например, *a*`[`*i*`]`, возвращается целое число, представляющее значение байта с индексом *i*. В операциях присваивания, таких как *a*`[`*i*`] = `*v*, значение *v* также должно быть целым 8-битным значением. Кроме того, тип `bytearray` поддерживает все операции, которые могут применяться к строкам (такие, как извлечение среза, `find()`, `split()`, `replace()` и так далее). В случае применения этих строковых операций все строковые литералы должны предваряться модификатором `b`, чтобы показать, что операции выполняются над строками байтов. Например, если требуется разбить массив байтов на поля, разделенные запятой, это следует делать так: `a.split(b','`), а не `a.split(',')`. Результатом таких операций всегда будет новый объект типа `bytearray`, а не строка. Чтобы преобразовать объект `bytearray` в строку, следует использовать метод *a*`.decode(`*encoding*`)`. При использовании кодировки `'latin-1'` в аргументе *encoding* объект типа `bytearray` будет преобразован в строку 8-битных символов без каких либо преобразований значений символов.

`bytearray(s ,encoding)`

Альтернативный способ создания экземпляра типа `bytearray` из строки символов *s*, где аргумент *encoding* определяет, какую кодировку символов использовать.

`bytes([x])`

Тип, используемый для представления неизменяемых массивов байтов. В Python 2 является псевдонимом функции `str()`, которая создает стандартную строку 8-битных символов. В Python 3 тип `bytes` является полностью самостоятельным типом, который является неизменяемой версией типа `bytearray`, описанного выше. В этом случае аргумент *x* интерпретируется точно так же и может использоваться тем же способом. Важное замечание, касающееся переносимости программ: даже при том, что в Python 2 определен тип `bytes`, его поведение отличается от поведения типа `bytes` в Python 3. Например, *a* – это экземпляр, созданный вызовом функции `bytes()`, тогда обращение *a*[*i*] в Python 2 вернет строку символов, а в Python 3 – целое число.

`bytes(s, encoding)`

Альтернативный способ создания экземпляра типа `bytes` из строки символов *s*, где аргумент *encoding* определяет, какую кодировку символов использовать. Только в Python 3.

`chr(x)`

Преобразует целое число *x* в строку из одного символа. В Python 2 аргумент *x* должен иметь значение в диапазоне `0 <= x <= 255`, а в Python 3 аргумент *x* должен быть представлен допустимым кодовым пунктом Юникода. Если значение *x* выходит за допустимый диапазон, возбуждается исключение `ValueError`.

`classmethod(func)`

Эта функция создает из функции *func* метод класса. Обычно используется только внутри определений классов, где она неявно вызывается декоратором `@classmethod`. В отличие от обычных методов, методы класса получают в первом аргументе ссылку на класс, а не на экземпляр. Например, если имеется объект `f`, который является экземпляром класса `Foo`, при вызове метода класса относительно объекта `f` в первом аргументе ему будет передана ссылка на класс `Foo`, а не на экземпляр `f`.

`cmp(x, y)`

Сравнивает *x* и *y* и возвращает отрицательное число, если `x < y`, положительное, если `x > y`, или `0`, если `x == y`. Сравниваться могут любые два объекта, однако результат может не иметь никакого смысла, если оба объекта не реализуют осмысленного метода сравнения (например, при сравнении числа и объекта файла). В некоторых случаях такая попытка сравнивания может привести к исключению.

`compile(string, filename, kind [, flags [, dont_inherit]])`

Компилирует строку *string* в объект программного кода для последующего использования с функцией `exec()` или `eval()`. Аргумент *string* – это строка

с допустимым программным кодом на языке Python. Если этот программный код занимает несколько текстовых строк, они должны завершаться одиночным символом перевода строки ('\n'), а не платформозависимым вариантом (например, '\r\n' в Windows). Аргумент *filename* – это строка с именем файла, в котором была определена строка *string*. Аргумент *kind* может иметь значение 'exec' – в случае последовательности инструкций, 'eval' – для одиночного выражения или 'single' – в случае единственной инструкции. Аргумент *flags* определяет, какие дополнительные особенности (включенные в модуль __future__) должны быть активированы. Эти особенности перечисляются в виде флагов, объявленных в модуле __future__, с помощью побитовой операции ИЛИ. Например, если требуется активировать новую семантику операции деления, следует установить в аргументе *flags* значение __future__.division.compiler_flag. Если аргумент *flags* опущен, он принимает значение по умолчанию 0, а программный код компилируется с учетом всех особенностей, действующих в текущей версии Python. Если функции передается аргумент *flags*, то особенности, перечисленные в нем, добавляются к списку действующих особенностей. Если аргумент *dont_inherit* установлен, активируются только особенности, перечисленные в аргументе *flags*, – особенности, действующие в текущей версии, игнорируются.

`complex([real [, imag]])`

Тип, используемый для представления комплексных чисел, состоящих из действительной и мнимой частей; аргументы *real* и *imag* могут иметь значения любых числовых типов. Если аргумент *imag* опущен, мнимая часть числа принимается равной нулю. Если в аргументе *real* передается строка, она анализируется и преобразуется в комплексное число. В этом случае аргумент *imag* не должен передаваться. Если ни один аргумент не задан, возвращается число 0j.

`delattr(object, attr)`

Удаляет атрибут *attr* объекта *object*. Аргумент *attr* – это строка. Функция действует точно так же, как и инструкция `del object.attr`.

`dict([m]) или dict(key1 = value1, key2 = value2, ...)`

Тип, используемый для представления словарей. При вызове без аргументов возвращает пустой словарь. Если в аргументе *m* передается объект отображения (такой как словарь), возвращается новый словарь с теми же ключами и значениями, что и в объекте *m*. Например, если *m* – это словарь, то вызов `dict(m)` просто создаст его поверхностную копию. Если аргумент *m* не является отображением, он должен поддерживать итерации, в ходе которых будет воспроизводиться последовательность пар (*key, value*). Эти пары будут использоваться для заполнения словаря. Функция `dict()` может также принимать именованные аргументы. Например, вызов `dict(foo=3, bar=7)` создаст словарь { 'foo' : 3, 'bar' : 7 }.

`dir([object])`

Возвращает отсортированный список имен атрибутов. Если аргумент *object* является модулем, функция вернет список имен, объявленных в этом модуле. Если аргумент *object* является объектом типа или класса, функ-

ция вернет список имен атрибутов. Имена обычно извлекаются из атрибута `__dict__` объекта, если он определен, но могут использоваться и другие источники. При вызове без аргумента возвращается список имен в текущей локальной таблице символов. Следует отметить, что главное назначение этой функции заключается в том, чтобы служить источником информации (например, для использования в интерактивном режиме). Она не может использоваться как официальный источник информации о программе, поскольку возвращаемые ею сведения могут быть неполными. Кроме того, пользовательские классы могут объявлять специальный метод `__dir__()`, способный влиять на результат этой функции.

`divmod(`*`a`*`, `*`b`*`)`

Возвращает частное и остаток от деления длинных целых чисел в виде кортежа. Для целых чисел возвращается значение (*a* `//` *b*, *a* `%` *b*). Для чисел с плавающей точкой — (`math.floor(`*a* `/` *b*`)`, *a* `%` *b*). Эта функция не должна вызываться для комплексных чисел.

`enumerate(`*`iter`*`[, `*`initial_value`*`)`

Для заданного итерируемого объекта *iter* возвращается новый итератор (типа `enumerate`), который воспроизводит последовательность кортежей, содержащих порядковый номер итерации и значение, полученное от объекта *iter*. Например, если допустить, что *iter* воспроизводит последовательность значений *a*, *b*, *c*, то итератор `enumerate(`*iter*`)` будет воспроизводить последовательность кортежей (0, *a*), (1, *b*), (2, *c*).

`eval(`*`expr`* `[, `*`globals`* `[, `*`locals`*`]])`

Вычисляет значение выражения *expr*. Аргумент *expr* — **это строка или объект** программного кода, созданный функцией `compile()`. Аргументы *globals* и *locals* — это объекты отображений, определяющие глобальное и локальное пространства имен соответственно. Если эти аргументы опущены, выражение вычисляется в пространстве имен вызывающего программного кода. В большинстве случаев в аргументах *globals* и *locals* передаются словари, но в сложных приложениях для этих целей могут использоваться собственные объекты отображений.

`exec(`*`code`* `[, `*`global`* `[, `*`locals`*`]])`

Выполняет инструкции языка Python. Аргумент *code* — это строка, файл или объект программного кода, созданный функцией `compile()`. Аргументы *globals* и *locals* определяют глобальное и локальное пространства имен соответственно. Если эти аргументы опущены, выполнение происходит в пространстве имен вызывающего программного кода. Если не задан один из аргументов *global* или *local*, поведение этой функции несколько отличается в разных версиях Python. В Python 2 `exec` в действительности реализована как специальная инструкция, тогда как в Python 3 она реализована как функция стандартной библиотеки. Такое различие в реализации порождает тонкий побочный эффект: в **Python 2 программный код, выполняемый** инструкцией `exec`, может изменять локальные переменные в пространстве имен вызывающего программного кода. В **Python 3 также имеется** возможность изменять переменные, но эти изменения не будут иметь эффекта за пределами самого вызова `exec()`. Это обусловлено тем, что в случае

отсутствия аргумента *locals* для получения локального пространства имен в Python 3 используется функция locals(). Как следует из описания функции locals(), возвращаемый ею словарь обеспечивает возможность чтения данных, но записанные в него данные не сохраняются.

filter(*function*, *iterable*)

В Python 2 эта функция создает список объектов, генерируемых итерируемым объектом *iterable*, для которых функция *function* возвращает значение True. В Python 3 возвращается итератор, который воспроизводит этот же результат. Если в аргументе *function* передается значение None, то каждый элемент в *iterable* проверяется на равенство значению False и те из них, для которых условие не выполняется, удаляются. Аргумент *iterable* может быть любым объектом, поддерживающим итерации. Как правило, фильтрацию можно выполнить гораздо быстрее с помощью выражений-генераторов или генераторов списков (глава 6).

float([*x*])

Тип, используемый для представления чисел с плавающей точкой. Если аргумент *x* является числом, оно преобразуется в число с плавающей точкой. Если аргумент *x* является строкой, она преобразуется в число с плавающей точкой. При вызове без аргумента возвращается число 0.0.

format(*value* [, *format_spec*])

Преобразует значение *value* в форматированную строку, в соответствии со спецификаторами формата в строке *format_spec*. Эта функция вызывает метод value.__format__(), который может интерпретировать спецификаторы формата по своему усмотрению. Для простых типов данных в число спецификаторов формата обычно входят символы выравнивания '<', '>' и '^'; числа (определяющие ширину поля); и символы кодов 'd', 'f' и 's' для целых чисел, для чисел с плавающей точкой и для строк соответственно. Например, спецификатор формата 'd' предназначен для форматирования целых чисел, спецификатор '8d' выровняет целое число по правому краю в поле шириной 8 символов, а спецификатор '<8d' выровняет целое число по левому краю в поле шириной 8 символов. Подробнее о функции format() и о спецификаторах формата рассказывается в главе 3 «Типы данных и объекты» и в главе 4 «Операторы и выражения».

frozenset([*items*])

Тип, используемый для представления неизменяемых объектов множеств, заполненных значениями, взятыми из аргумента *items*, который должен быть итерируемым объектом. Значения также должны быть неизменяемыми. При вызове без аргумента возвращается пустое множество.

getattr(*object*, *name* [,*default*])

Возвращает значение атрибута *name* объекта *object*. Аргумент *name* должен быть строкой, содержащей имя атрибута. Необязательный аргумент *default* определяет значение, возвращаемое в случае отсутствия атрибута. Если этот аргумент не задан, возбуждается исключение AttributeError. Возвращает тот же результат, что и выражение *object.name*.

`globals()`

Возвращает словарь текущего модуля, который представляет глобальное пространство имен. При вызове из функции или метода возвращает глобальное пространство имен для модуля, в котором была определена эта функция или метод.

`hasattr(object, name)`

Возвращает `True`, аргумент *`name`* является именем атрибута объекта *`object`*. В противном случае возвращается значение `False`. Аргумент *`name`* должен быть строкой.

`hash(object)`

Возвращает целочисленное значение хеша для объекта *`object`* (если это возможно). Значения хешей в первую очередь используются в реализациях словарей, множеств и других объектов отображений. Два объекта, которые признаются равными, имеют одинаковые значения хешей. Изменяемые объекты не поддерживают возможность вычисления хеша, однако пользовательские объекты могут определять метод `__hash__()`, чтобы обеспечить поддержку этой операции.

`help([object])`

Обращается к справочной системе во время интерактивных сеансов. Аргумент *`object`* может быть строкой с именем модуля, класса, функции, метода, с ключевым словом или названием раздела в документации. Если передается объект какого-либо другого типа, будет воспроизведена справочная информация для этого объекта. При вызове без аргумента будет запущен инструмент предоставления интерактивной справки с дополнительной информацией.

`hex(x)`

Вернет строку с шестнадцатеричным представлением целого числа *`x`*.

`id(object)`

Возвращает уникальный целочисленный идентификатор объекта *`object`*. Этому значению не следует придавать какой-либо особый смысл (считать, например, что это адрес объекта в памяти).

`input([prompt])`

В Python 2 эта функция выводит приглашение к вводу *`prompt`*, читает введенную строку и обрабатывает ее с помощью функции `eval()` (то есть это то же самое, что и вызов `eval(raw_input(`*`prompt`*`))`). В Python 3 приглашение к вводу *`prompt`* выводится в поток стандартного вывода, а введенная строка возвращается без какой-либо дополнительной обработки.

`int(x [,base])`

Тип, используемый для представления целых чисел. Если аргумент *`x`* является числом, оно будет преобразовано в целое число, усечением дробной части до 0. Если это строка, она будет преобразована в целочисленное значение. Необязательный аргумент *`base`* определяет основание системы

счисления и используется при преобразовании строки в число. В Python 2 длинные целые числа создаются, только когда значение *x* не умещается в 32-битный тип int.

`isinstance(object, classobj)`

Возвращает True, если объект *object* является экземпляром класса *classobj*, или подкласса класса *classobj*, или абстрактного базового класса *classobj*. Аргумент *classobj* может также быть кортежем возможных типов или классов. Например, вызов isinstance(*s*, (list,tuple)) вернет True, если объект *s* является кортежем или списком.

`issubclass(class1, class2)`

Возвращает True, если класс *class1* является подклассом класса (то есть наследует) *class2* или если класс *class1* зарегистрирован в абстрактном базовом классе *class2*. Аргумент *class2* может также быть кортежем возможных классов, в этом случае будет проверен каждый класс в кортеже. Обратите внимание, что вызов issubclass(*A*, *A*) вернет True.

`iter(object [,sentinel])`

Возвращает итератор, воспроизводящий элементы объекта *object*. Если аргумент *sentinel* опущен, объект *object* должен реализовать либо метод __iter__(), который создает итератор, либо метод __getitem__(), который принимает целочисленные аргументы со значениями, начиная с 0. При наличии аргумента *sentinel* аргумент *object* интерпретируется иначе. В этом случае объект *object* должен быть вызываемым объектом, не принимающим аргументов. Возвращаемый им итератор будет вызываться этой функцией в цикле до тех пор, пока он не вернет значение, равное значению аргумента *sentinel*, после чего итерации будут остановлены. Если объект *object* не поддерживает итерации, будет возбуждено исключение TypeError.

`len(s)`

Возвращает количество элементов, содержащихся в *s*. Аргумент *s* должен быть списком, кортежем, строкой, множеством или словарем. Если аргумент *s* является итерируемым объектом другого типа, например генератором, возбуждается исключение TypeError.

`list([items])`

Тип, используемый для представления списков. Аргумент *items* может быть любым итерируемым объектом, значения которого будут использоваться для заполнения списка. Если аргумент *items* уже является списком, создается его копия. При вызове без аргумента возвращается пустой список.

`locals()`

Возвращает словарь, соответствующий локальному пространству имен вызывающей функции. Этот словарь должен использоваться только для получения значений – изменения, выполненные в этом словаре, не сохраняются в локальном пространстве имен.

`long([`*x* `[, ` *base*`]])`

Тип, используемый для представления длинных целых чисел в Python 2. Если аргумент *x* является числом, оно будет преобразовано в целое число усечением дробной части до 0. Если это строка, она будет преобразована в длинное целое число. При вызове без аргументов возвращается значение `0L`. Для обеспечения переносимости следует избегать непосредственного использования функции `long()`. Функция `int(`*x*`)` будет создавать длинные целые числа в случае необходимости. Чтобы убедиться, что аргумент *x* является целым числом, проверку типа следует проводить следующим образом: `isinstance(x, numbers.Integral)`.

`map(`*function*`, ` *items*`, ...)`

В Python 2 применяет функцию *function* к каждому элементу последовательности *items* и возвращает список результатов. В Python 3 создается итератор, который воспроизводит те же результаты. Если функции `map()` передается несколько последовательностей, это предполагает, что количество аргументов, принимаемых функцией *function*, совпадает с количеством последовательностей, когда каждый аргумент берется из другой последовательности. В случае нескольких последовательностей поведение `map()` в Python 2 и Python 3 отличается. В Python 2 результат будет иметь ту же длину, что и самая длинная последовательность, а в случае исчерпания более коротких последовательностей вместо недостающих значений будет использоваться `None`. В Python 3 результат будет иметь ту же длину, что и самая короткая последовательность. Вместо функции `map()` практически всегда лучше использовать выражения-генераторы или генераторы списков (которые обладают более высокой производительностью). Например, вызов `map(`*function*`, ` *s*`)` обычно можно заменить генератором списков `[`*function*`(`*x*`) for` *x* `in` *s*`]`.

`max(`*s* `[, ` *args*`, ...])`

Если передается единственный аргумент *s*, возвращает элемент последовательности *s* с максимальным значением. Аргумент *s* может быть любым итерируемым объектом. Если передается несколько аргументов, функция возвращает наибольший из аргументов.

`min(`*s* `[, ` *args*`, ...])`

Если передается единственный аргумент *s*, возвращает элемент последовательности *s* с минимальным значением. Аргумент *s* может быть любым итерируемым объектом. Если передается несколько аргументов, функция возвращает наименьший из аргументов.

`next(`*s* `[, ` *default*`])`

Возвращает следующий элемент итератора *s*. После того как все элементы итератора будут исчерпаны, возбуждает исключение `StopIteration`, если не указано значение по умолчанию в аргументе *default*. В противном случае возвращается значение по умолчанию. Для обеспечения переносимости вместо непосредственного вызова метода *s*`.next()` итератора *s* всегда следует использовать эту функцию. В Python 3 соответствующий метод итераторов

был переименован в `s.__next__()`. Если в программном коде использовать только функцию `next()`, вам не придется беспокоиться об этих различиях.

`object()`

Базовый класс всех объектов в языке Python. Его можно использовать для создания экземпляров, но результат не представляет особого интереса.

`oct(x)`

Вернет строку с восьмеричным представлением целого числа *x*.

`open(filename [, mode [, bufsize]])`

В Python 2 открывает файл *filename* **и возвращает новый объект файла (глава 9 «Ввод и вывод»)**. Аргумент *mode* – это строка, описывающая режим открытия файла: `'r'` – для чтения, `'w'` – для записи и `'a'` – для добавления в конец. Второй символ строки, `'t'` или `'b'`, используется для определения текстового (по умолчанию) или двоичного режима. Например, если указана строка режима `'r'` или `'rt'`, файл будет открыт в текстовом режиме, а если указана строка `'rb'`, файл будет открыт в двоичном режиме. В строку режима может быть добавлен необязательный символ `'+'`, чтобы указать, что файл должен быть доступен для обновления (то есть допускаются операции чтения и записи). При выборе режима `'w+'` размер существующего файла будет усечен до нуля. При выборе режима `'r+'` или `'a+'` файл будет открыт как для чтения, так и для записи, но при этом первоначальное содержимое файла останется нетронутым. При выборе режима `'U'` или `'rU'` файл будет открыт в режиме поддержки универсального символа перевода строки. В этом режиме все возможные варианты символов завершения строки (`'\n'`, `'\r'`, `'\r\n'`) будут преобразовываться в стандартный символ `'\n'`. Если аргумент *mode* опущен, предполагается режим `'rt'`. Аргумент *bufsize* определяет поведение механизма буферизации, где значение `0` означает отсутствие буферизации, `1` – построчную буферизацию, а любое другое значение интерпретируется как приблизительный размер буфера в байтах. Отрицательное число указывает, что должны использоваться системные параметры настройки буферизации (по умолчанию).

`open(filename[, mode [, bufsize [, encoding [, errors [, newline [, closefd]]]]]])`

В Python 3 открывает файл *filename* и возвращает новый объект файла. Первые три аргумента имеют то же значение, что и в версии **Python 2, описанной выше**. Аргумент *encoding* – это название кодировки символов, такое как `'utf-8'`. Аргумент *errors* определяет политику обработки ошибок кодирования символов и может иметь одно из следующих значений: `'strict'`, `'ignore'`, `'replace'`, `'backslashreplace'` **или** `'xmlcharrefreplace'`. **Аргумент** *newline* определяет поведение режима поддержки универсального символа перевода строки и может иметь значение `None`, `''`, `'\n'`, `'\r'` или `'\r\n'`. Аргумент *closefd* – это логический флаг, который определяет, должен ли фактически закрываться дескриптор файла при вызове метода `close()`. В отличие от версии в Python 2, функция `open()` в Python 3 может возвращать объекты, обладающие различными возможностями, в зависимости от выбранного режима ввода-вывода. Например, если файл открывается в двоичном режиме, возвращается объект, который выполняет операции ввода-вывода, такие

как `read()` и `write()`, не над строками, а над массивами байтов. Файловые операции ввода-вывода – это одна из областей, где имеются существенные различия между Python 2 и 3. Дополнительные подробности приводятся в приложении А, «Python 3».

`ord(c)`

Возвращает целочисленный код символа *c*. Для обычных символов возвращается значение в диапазоне `[0,255]`. Для символов Юникода обычно возвращается значение в диапазоне `[0,65535]`. В Python 3 аргумент *c* может быть суррогатной парой Юникода, в этом случае она будет преобразована в кодовый пункт соответствующего символа Юникода.

`pow(x, y [, z])`

Возвращает результат выражения `x ** y`. Если аргумент *z* определен, возвращается результат выражения `(x ** y) % z`. Если заданы все три аргумента, все они должны быть целыми числами, а аргумент *y* в этом случае должен быть неотрицательным числом.

`print(value, ... [, sep=separator, end=ending, file=outfile])`

Функция в **Python 3, используемая для вывода последовательности значений**. Может принимать любое число значений, при этом все они будут выводиться в одной строке. Именованный аргумент *sep* позволяет определить свой символ-разделитель (по умолчанию используется пробел). Именованный аргумент *end* позволяет определить свой символ завершения строки (по умолчанию используется `'\n'`). Именованный аргумент *file* позволяет перенаправить вывод в объект файла. Эта функция может использоваться в Python 2, если добавить в программный код инструкцию `from __future__ import print_function`.

`property([fget [,fset [,fdel [,doc]]]])`

Создает атрибут-свойство класса. Аргумент *fget* – это функция, возвращающая значение атрибута, *fset* – устанавливает новое значение атрибута, а *fdel* – удаляет атрибут. Аргумент *doc* **содержит строку документирования. Все аргументы могут передаваться как именованные аргументы, например** `property(fget=getX, doc="some text")`.

`range([start,] stop [, step])`

В Python 2 создает список, заполненный целыми числами в диапазоне от *start* до *stop*. Аргумент *step* позволяет определить шаг и по умолчанию имеет значение `1`. Если аргумент *start* опущен (когда функция `range()` вызывается с одним аргументом), по умолчанию он принимается равным `0`. При отрицательном значении аргумента *step* создается список с числами в порядке убывания.[1] В Python 3 функция `range()` создает специальный объект `range`, который вычисляет значения по требованию (как это делает функция `xrange()` в предыдущих версиях Python).

[1] В этом случае значение аргумента *start* должно быть больше значения аргумента *stop*. – *Прим. перев.*

`raw_input([prompt])`

Функция в Python 2, которая читает данные из потока стандартного ввода (`sys.stdin`) и возвращает их в виде строки. Если аргумент *prompt* указан, он будет выведен в поток стандартного вывода (`sys.stdout`) перед чтением ввода. Завершающие символы перевода строки удаляются, а если будет прочитан признак конца файла EOF, возбуждается исключение `EOFError`. Если загружен модуль `readline`, эта функция будет использовать его, чтобы обеспечить дополнительные возможности редактирования строки ввода и автоматического завершения. Для организации чтения ввода в Python 3 вместо этой функции следует использовать функцию `input()`.

`repr(object)`

Возвращает строковое представление объекта *object*. В большинстве случаев возвращаемая строка содержит выражение, с помощью которого можно воспроизвести объект, если передать ее функции `eval()`. Следует помнить, что в Python 3 результатом этой функции может быть строка Юникода, которая не может быть отображена в терминале или в окне командной оболочки (вызывая исключение). Чтобы создать строковое представление объекта *object* в кодировке ASCII, можно воспользоваться функцией `ascii()`.

`reversed(s)`

Создает итератор для обхода последовательности *s* в обратном порядке. Эта функция может работать только с последовательностями, реализующими методы `__len__()` и `__getitem__()`. Кроме того, индексы элементов *s* должны начинаться с `0`. Функция не может применяться к генераторам и итераторам.

`round(x [, n])`

Округляет число *x* с плавающей точкой до ближайшего кратного «10 в степени минус *n*». Если аргумент *n* опущен, используется значение по умолчанию `0`. Если число *x* оказывается одинаково близким к двум возможным результатам, в Python 2 округление будет выполнено до числа, дальше отстоящего от 0 (например, число `0.5` будет округлено до `1.0`, а число `-0.5` — до `-1.0`). В Python 3 округление будет выполнено до числа, стоящего ближе к 0, если предыдущая цифра четная, и до числа, дальше отстоящего от 0, в противном случае (например, число `0.5` будет округлено до `0.0`, число `1.5` будет округлено до `2`).

`set([items])`

Создает множество, заполненное элементами итерируемого объекта *items*. Аргумент *items* должен быть неизменяемым. Если аргумент *items* содержит другие множества, эти множества должны иметь тип `frozenset`. При вызове без аргумента возвращается пустое множество.

`setattr(object, name, value)`

Создает в объекте *object* атрибут *name* и записывает в него значение *value*. Аргумент *name* должен быть строкой. Выполняет ту же операцию, что и инструкция *object.name = value*.

`slice([start,] stop [, step])`

Возвращает объект среза, представленный указанными целыми числами. Кроме того, объекты срезов создаются с использованием расширенного синтаксиса срезов `a[i:j:k]`. Дополнительные подробности о срезах приводятся в разделе «Методы последовательностей и отображений» в главе 3.

`sorted(iterable [, key=keyfunc [, reverse=reverseflag]])`

Создает отсортированный список из элементов итерируемого объекта `iterable`. В именованном аргументе `key` передается функция, принимающая единственный аргумент, которая используется для преобразования значений, прежде чем они будут переданы функции сравнивания. В именованном аргументе `reverse` передается логический флаг, определяющий, должен ли список сортироваться в обратном порядке. Аргументы `key` и `reverse` должны передаваться как именованные аргументы, например `sorted(a,key=get_name)`.

`staticmethod(func)`

Создает статический метод класса. Эта функция неявно вызывается декоратором `@staticmethod`.

`str([object])`

Тип, используемый для представления строк. В Python 2 строки состоят из 8-битных символов, тогда как в Python 3 строки состоят из символов Юникода. Если функции передается аргумент `object`, создается строковое представление этого объекта, для чего вызывается его метод `__str__()`. Это та строка, которая выводится инструкцией `print`. При вызове без аргумента возвращается пустая строка.

`sum(items [,initial])`

Вычисляет сумму последовательности элементов итерируемого объекта `items`. В аргументе `initial` может передаваться начальное значение, которое по умолчанию равно 0. Эта функция работает только с числами.

`super(type [, object])`

Возвращает объект, представляющий суперклассы типа `type`. Чаще всего возвращаемый объект используется для вызова методов базового класса. Например:

```
class B(A):
    def foo(self):
        super(B,self).foo()
```

Если в аргументе `object` передается объект, для него вызов функции `isinstance(object, type)` должен возвращать `True`. Если в аргументе `object` передается тип, он должен быть подклассом класса `type`. Дополнительные подробности приводятся в главе 7 «Классы и объектно-ориентированное программирование». В Python 3 допускается использовать функцию `super()` без аргументов, при вызове из методов. В этом случае в аргументе `type` по умолчанию передается класс, в котором был определен метод, а в аргументе `object` передается первый аргумент метода. Хотя синтаксис стал

более очевидным, тем не менее функция утратила обратную совместимость с Python 2, поэтому, если проблема переносимости программного кода имеет большое значение, такой синтаксис лучше не использовать.

`tuple([items])`

Тип, используемый для представления кортежей. Аргумент *items* должен быть итерируемым объектом, который будет использоваться для заполнения кортежа. Однако если *items* уже является кортежем, функция просто возвращает его без каких-либо изменений. При вызове без аргумента возвращается пустой кортеж.

`type(object)`

Базовый класс для всех типов в языке Python. Когда вызывается как функция, возвращает тип объекта *object*. Возвращаемый тип – это класс объекта. Для обычных типов, таких как целые числа, числа с плавающей точкой и списков, возвращаемый тип будет ссылаться на один из встроенных классов, таких как `int`, `float`, `list` и так далее. Для пользовательских объектов типом является соответствующий класс. Для объектов, связанных с внутренней реализацией Python, обычно возвращается ссылка на один из классов, объявленных в модуле `types`.

`type(name, bases, dict)`

Создает новый объект *type* (как при объявлении нового класса). В аргументе *name* передается имя типа, в аргументе *bases* – кортеж базовых классов, а в аргументе *dict* – словарь, содержащий определения, соответствующие телу класса. Чаще всего эта функция используется при работе с метаклассами. Подробнее об этом рассказывается в главе 7.

`unichr(x)`

Преобразует целое или длинное целое число *x*, где `0 <= x <= 65535`, в символ Юникода. Функция определена только в Python 2. В Python 3 можно просто использовать функцию `chr(x)`.

`unicode(string [,encoding [,errors]])`

Функция определена только в Python 2. Преобразует обычную строку в строку Юникода. Аргумент *encoding* определяет кодировку строки. Если этот аргумент опущен, по умолчанию используется кодировка, возвращаемая функцией `sys.getdefaultencoding()`. Аргумент *errors* определяет политику обработки ошибок кодирования символов и может иметь одно из следующих значений: `'strict'`, `'ignore'`, `'replace'`, `'backslashreplace'` или `'xmlcharrefreplace'`. Дополнительная информация приводится в главе 9 и в главе 3. Недоступна в Python 3.

`vars([object])`

Возвращает таблицу символов объекта *object* (обычно хранится в атрибуте `__dict__`). При вызове без аргумента возвращается словарь, соответствующий локальному пространству имен. Словарь, возвращаемый этой функцией, должен рассматриваться как доступный только для чтения. Изменения в этом словаре не сохраняются в локальном пространстве имен или в таблице символов объекта.

`xrange([`*start*`,] `*stop*` [, `*step*`])`

Тип, используемый для представления диапазонов целых чисел от *start* до *stop*, причем значение *stop* не входит в этот диапазон. В необязательном аргументе *step* передается шаг. Значения фактически нигде не сохраняются, а вычисляются по требованию, при попытке обращения. В Python 2 функция `xrange()` является предпочтительным способом определения диапазонов целочисленных значений для использования в циклах. В Python 3 функция `xrange()` была переименована в `range()`, а функция `xrange()` была ликвидирована. Значения аргументов *start*, *stop* и *step* ограничены множеством значений целых чисел, допустимых для аппаратной платформы (обычно 32 бита).

`zip([`*s1* `[, `*s2* `[,..]]])`

В Python 2 возвращает список кортежей, где *n*-й кортеж имеет вид (*s1*`[n], `*s2*`[n], ...`). Длина возвращаемого списка равна длине самой короткой последовательности, полученной в виде аргумента. При вызове без аргументов возвращается пустой список. В Python 3 функция действует похожим образом, но в качестве результата возвращается итератор, который воспроизводит последовательность кортежей. В Python 2 использование функции `zip()` с длинными входными последовательностями может приводить к потреблению значительных объемов памяти. Чтобы избежать этого, вместо нее можно использовать функцию `itertools.izip()`.

Встроенные исключения

Встроенные исключения объявляются в модуле `exceptions`, который всегда загружается перед запуском любой программы. Исключения объявлены как классы.

Базовые классы исключений

Следующие исключения играют роль базовых классов для всех остальных исключений:

`BaseException`

Базовый класс для всех исключений. Все встроенные исключения являются производными этого класса.

`Exception`

Базовый класс для всех программных исключений, включая все встроенные исключения, кроме `SystemExit`, `GeneratorExit` и `KeyboardInterrupt`. Пользовательские исключения должны наследовать класс `Exception`.

`ArithmeticError`

Базовый класс для исключений, возбуждаемых арифметическими операциями, включая `OverflowError`, `ZeroDivisionError` и `FloatingPointError`.

`LookupError`

Базовый класс исключений, возникающих при обращении к недопустимому индексу или ключу, включая `IndexError` и `KeyError`.

`EnvironmentError`

Базовый класс для ошибок, возникающих за пределами интерпретатора Python, включая `IOError` и `OSError`.

Предыдущие исключения никогда не возбуждаются явно. Однако они могут использоваться для обработки определенных классов ошибок. Например, следующий фрагмент будет обрабатывать любые ошибки арифметических операций:

```
try:
    # Некоторая операция
    ...
except ArithmeticError as e:
    # Обработка арифметической ошибки
```

Экземпляры исключений

Когда возбуждается какое-либо исключение, создается экземпляр класса исключения. Этот экземпляр сохраняется в необязательной переменной, указанной в инструкции `except`. Например:

```
except IOError as e:
    # Обработать ошибку
    # 'e' - это экземпляр класса IOError
```

Экземпляры *e* исключений имеют ряд стандартных атрибутов, которые могут использоваться для определенных целей.

e.`args`

Кортеж аргументов, полученных конструктором при возбуждении исключения. В большинстве случаев этот кортеж содержит единственный элемент – строку с текстом сообщения об ошибке. Для исключений класса `EnvironmentError` значением этого атрибута может быть кортеж из 2 или из 3 элементов, содержащий целочисленный код ошибки, строку с текстом сообщения об ошибке и, возможно, имя файла. Содержимое этого кортежа можно использовать для повторного возбуждения исключения в другом контексте; например, в другом процессе интерпретатора Python.

e.`message`

Строка с текстом сообщения, которое выводится при отображении информации об исключении (только в Python 2).

e.`__cause__`

Предыдущее исключение в явной цепочке исключений (только в Python 3). Подробнее об этом рассказывается в приложении А.

e.`__context__`

Предыдущее исключение в неявной цепочке исключений (только в Python 3). Подробнее об этом рассказывается в приложении А.

e.`__traceback__`

Объект с трассировочной информацией, ассоциированный с исключением (только в Python 3). Подробнее об этом рассказывается в приложении А.

Предопределенные классы исключений

Ниже приводится список типов исключений, возбуждаемых программами:

`AssertionError`

Не выполнено условие в инструкции `assert`.

`AttributeError`

Ошибка при попытке обращения к атрибуту или при попытке присвоить ему значение.

`EOFError`

Конец файла. Возбуждается встроенными функциями `input()` и `raw_input()`. Следует отметить, что большинство других операций ввода-вывода, таких как методы `read()` и `readline()` объектов файлов, по достижении конца файла не возбуждают исключение, а возвращают пустую строку.

`FloatingPointError`

Ошибка выполнения операции над числами с плавающей точкой. Следует отметить, что обработка исключений, возникающих в операциях над числами с плавающей точкой, является достаточно сложной проблемой, так как это единственное исключение, которое возбуждается, только если интерпретатор Python был настроен и скомпилирован с его поддержкой. Чаще подобные ошибки не приводят к исключению, а воспроизводят результат, такой как `float('nan')` или `float('inf')`. Является производным от класса `ArithmeticError`.

`GeneratorExit`

Возбуждается внутри функций-генераторов как сигнал о завершении работы. Это происходит при преждевременном уничтожении генератора (до того, как будут исчерпаны все значения генератора) или при вызове метода `close()` генератора. Если генератор проигнорирует это исключение, сам генератор будет уничтожен, а исключение просто проигнорировано.

`IOError`

Ошибка операции ввода-вывода. В результате ошибки создается экземпляр `IOError` с атрибутами `errno`, `strerror` и `filename`, где `errno` – целочисленный код ошибки, `strerror` – строка с текстом сообщения об ошибке и `filename` – имя файла. Является производным от класса `EnvironmentError`.

`ImportError`

Возбуждается, когда инструкция `import` не может отыскать модуль или когда инструкция `from` не может отыскать указанное имя в модуле.

`IndentationError`

Ошибка оформления отступов. Является производным от класса `SyntaxError`.

`IndexError`

Выход индекса за пределы последовательности. Является производным от класса `LookupError`.

`KeyError`

Требуемый ключ не найден в объекте отображения. Является производным от класса `LookupError`.

`KeyboardInterrupt`

Возбуждается, когда пользователь нажимает клавишу прерывания программы (обычно это комбинация `Ctrl+C`).

`MemoryError`

Ошибка нехватки памяти, допускающая возможность восстановления.

`NameError`

Требуемое имя не найдено в локальном или глобальном пространстве имен.

`NotImplementedError`

Нереализованная особенность. Может возбуждаться базовыми классами, когда происходит обращение к методу, который должен быть реализован в производных классах. Является производным от класса `RuntimeError`.

`OSError`

Ошибка операционной системы. В основном возбуждается функциями в модуле `os`. В результате ошибки создается такой же экземпляр, как и в случае с исключением `IOError`. Является производным от класса `EnvironmentError`.

`OverflowError`

Возбуждается, когда в результате выполнения некоторой операции получается слишком большое целое число. Это исключение обычно возникает при передаче больших целочисленных значений объектам, реализация которых опирается на аппаратное представление целых чисел фиксированного размера. Например, эта ошибка может возникнуть в объектах `range` или `xrange`, если передать им начальное или конечное значение, не умещающееся в 32-битное представление. Является производным от класса `ArithmeticError`.

`ReferenceError`

Возникает при попытке обращения к объекту по слабой ссылке, после того как сам объект уже был уничтожен. Дополнительную информацию можно найти в документации к модулю `weakref`.

`RuntimeError`

Ошибка, не подпадающая ни под одну другую категорию.

`StopIteration`

Возбуждается, чтобы известить об окончании итераций. Это обычно происходит в методе `next()` объекта или внутри функции-генератора.

`SyntaxError`

Синтаксическая ошибка. Экземпляры этого исключения имеют атрибуты `filename`, `lineno`, `offset` и `text`, которые могут использоваться для получения дополнительной информации.

`SystemError`

Внутренняя ошибка интерпретатора. Значением этого исключения является строка с текстом, описывающим проблему.

`SystemExit`

Возбуждается функцией `sys.exit()`. Значением этого исключения является целое число – код завершения. Для немедленного завершения программы можно использовать функцию `os._exit()`.

`TabError`

Непоследовательное использование символов табуляции. Возбуждается, когда интерпретатор Python запущен с ключом `-tt`. Является производным от класса `SyntaxError`.

`TypeError`

Возбуждается, когда оператор или функция применяется к объекту несовместимого типа.

`UnboundLocalError`

Попытка обращения к несвязанной локальной переменной. Эта ошибка возникает при попытке обратиться к переменной до того, как она будет определена в функции. Является производным от класса `NameError`.

`UnicodeError`

Ошибка кодирования или декодирования символа Юникода. Является производным от класса `ValueError`.

`UnicodeEncodeError`

Ошибка кодирования символа Юникода. Является производным от класса `UnicodeError`.

`UnicodeDecodeError`

Ошибка декодирования символа Юникода. Является производным от класса `UnicodeError`.

`UnicodeTranslateError`

Ошибка при попытке преобразования символа Юникода. Является производным от класса `UnicodeError`.

`ValueError`

Возбуждается, когда аргумент функции или операции имеет совместимый тип, но недопустимое значение.

`WindowsError`

Возбуждается в случае ошибки при обращении к системным вызовам в Windows. Является производным от класса `OSError`.

`ZeroDivisionError`

Деление на ноль. Является производным от класса `ArithmeticError`.

Встроенные предупреждения

В состав стандартной библиотеки Python входит модуль `warnings`, который обычно используется, чтобы сообщить программистам об использовании нерекомендуемых возможностей. Предупреждения возбуждаются включением программного кода, как показано ниже:

```
import warnings
warnings.warn("Флаг MONDO больше не поддерживается", DeprecationWarning)
```

Несмотря на то что предупреждения возбуждаются средствами библиотечного модуля, тем не менее имена различных предупреждений являются встроенными. Предупреждения до определенной степени напоминают исключения. Иерархия встроенных предупреждений наследует класс `Exception`.

`Warning`

Базовый класс для всех предупреждений. Является производным от класса `Exception`.

`UserWarning`

Универсальное предупреждение, определяемое пользователем. Является производным от класса `Warning`.

`DeprecationWarning`

Предупреждение на случай использования нерекомендуемых возможностей. Является производным от класса `Warning`.

`SyntaxWarning`

Предупреждение на случай использования нерекомендуемого синтаксиса. Является производным от класса `Warning`.

`RuntimeWarning`

Предупреждение на случай использования особенностей, которые могут привести к проблемам во время выполнения. Является производным от класса `Warning`.

`FutureWarning`

Предупреждение о том, что поведение используемой возможности изменится в будущем. Является производным от класса `Warning`.

Предупреждения отличаются от исключений тем, что возбуждение предупреждения с помощью функции `warn()` может приводить, а может не приводить к остановке программы. Например, предупреждение может просто вызывать вывод какой-либо информации или приводить к исключению. Фактическое поведение предупреждений может быть настроено средствами модуля `warnings` или с помощью ключа `-W` интерпретатора. Если в программе используется какой-либо программный код, генерирующий предупреждение, после появления которого было бы желательно продолжить работу, предупреждение, преобразованное в исключение, можно перехватить с помощью инструкций `try` и `except`. Например:

```
try:
    import md5
except DeprecationWarning:
    pass
```

Следует отметить, что такой способ используется достаточно редко. Хотя он позволяет перехватывать предупреждения, преобразованные в исключения, тем не менее он не подавляет вывод сообщений (для управления выводом предупреждений необходимо использовать модуль `warnings`). Кроме того, игнорирование предупреждений – это отличный способ написать программу, которая будет работать с ошибками после выхода новых версий Python.

Модуль future_builtins

Модуль `future_builtins`, доступный только в Python 2, содержит реализации встроенных функций, поведение которых изменилось в Python 3. В модуле определены следующие функции:

`ascii(object)`

Воспроизводит тот же результат, что и функция `repr()`. Описание приводится в разделе «Встроенные функции» в этой главе.

`filter(`*`function, iterable`*`)`

Создает итератор вместо списка. То же, что и функция `itertools.ifilter()`.

`hex(`*`object`*`)`

Создает строку с шестнадцатеричным представлением целого числа, но для получения целочисленного значения вместо метода `__hex__()` использует специальный метод `__index__()`.

`map(`*`function, iterable, ...`*`)`

Создает итератор вместо списка. То же, что и функция `itertools.imap()`.

`oct(`*`object`*`)`

Создает строку с восьмеричным представлением целого числа, но для получения целочисленного значения вместо метода `__oct__()` использует специальный метод `__index__()`.

`zip(`*`iterable, iterable, ...`*` )`

Создает итератор вместо списка. То же, что и функция `itertools.izip()`.

Не забывайте, что список функций в этом модуле не является исчерпывающим списком изменений в модуле `builtins`. Например, в Python 3 функция `raw_input()` была переименована в `input()`, а функция `xrange()` – в `range()`.

13

Службы Python времени выполнения

В этой главе описываются модули, связанные с особенностями работы интерпретатора Python. Здесь рассматриваются такие темы, как сборка мусора, основы управления объектами (копирование, сериализация и так далее), слабые ссылки и окружение интерпретатора.

Модуль atexit

Модуль `atexit` используется для регистрации функций, которые должны вызываться интерпретатором перед завершением работы. Он предоставляет единственную функцию:

`register(`*`func`* `[,`*`args`* `[,`*`kwargs`*`]])`

Добавляет функцию *`func`* в список функций, которые должны вызываться интерпретатором перед завершением работы. *`args`* – кортеж аргументов, которые будут переданы функции *`func`*. *`kwargs`* – словарь именованных аргументов. Функция будет вызвана как `func(`*`*args,**kwargs`*`)`. Функции будут вызываться в порядке, обратном порядку регистрации (функция, зарегистрированная последней, будет вызвана первой). Если в процессе выполнения функции возникнет исключение, сообщение будет выведено в поток стандартного вывода, но само исключение будет проигнорировано.

Модуль copy

Модуль `copy` предоставляет функции создания поверхностных и глубоких копий составных объектов, включая списки, кортежи, словари и экземпляры пользовательских объектов.

`copy(`*`x`*`)`

Создает поверхностную копию объекта *`x`* за счет создания нового составного объекта и копирования элементов объекта *`x`* по ссылке. Для объектов встроенных типов эту функцию использовать не принято. Вместо этого обычно вызываются функции `list(`*`x`*`)`, `dict(`*`x`*`)`, `set(`*`x`*`)` и так далее, которые

создают поверхностные копии аргумента *x* (следует отметить, что прямые обращения к именам типов, подобно приведенным, кроме всего прочего, выполняются значительно быстрее, чем функция copy()).

deepcopy(*x* [, *visit*])

Создает глубокую копию объекта *x* за счет создания нового составного объекта и рекурсивного копирования всех атрибутов объекта *x*. Необязательный аргумент *visit* – это словарь, который используется для запоминания посещенных объектов, чтобы избежать зацикливания при копировании рекурсивных структур данных. Этот аргумент обычно указывается, только если deepcopy() вызывается рекурсивно, как описано ниже в этой главе.

Хотя обычно это не требуется, тем не менее класс может реализовать собственные версии методов копирования, в виде методов __copy__(*self*) и __deepcopy__(*self*, *visit*) – для поверхностного и глубокого копирования соответственно. Метод __deepcopy__() должен принимать словарь *visit*, который служит для запоминания объектов, встретившихся в процессе копирования. Все, что метод __deepcopy__() должен сделать со словарем *visit*, – это передать его другим вызовам функции deepcopy(), присутствующим в реализации метода (если таковые имеются).

Если класс реализует методы __getstate__() и __setstate__(), которые используются модулем pickle, для создания копий модулем copy будут использоваться эти методы.

Примечания

- Этот модуль может использоваться для работы с простыми типами, такими как целые числа и строки, однако в этом нет никакой необходимости.
- Функции копирования не могут применяться к модулям, объектам классов, функциям, методам, к объектам с трассировочной информацией, кадрам стека, файлам, сокетам и к другим подобным типам. Когда объект не может быть скопирован, возбуждается исключение copy.error.

Модуль gc

Модуль gc предоставляет интерфейс управления сборщиком мусора, способным обнаруживать циклические ссылки в объектах, таких как списки, кортежи, словари, и в экземплярах других классов. При создании объектов различных контейнерных типов они заносятся во внутренний список интерпретатора. Всякий раз, когда контейнерный объект уничтожается, он удаляется из этого списка. Когда количество операций создания новых объектов превысит количество удалений объектов на некоторое пороговое значение, определяемое пользователем, запускается сборщик мусора. Сборщик мусора просматривает этот список и выявляет коллекции объектов, которые больше не используются, но еще не были удалены из-за наличия циклических ссылок. Кроме того, сборщик мусора использует трехуровневую схему сборки мусора разных поколений, когда объекты, пережившие первый этап сборки мусора, помещаются в списки объектов, которые про-

веряются реже. Благодаря этому обеспечивается более высокая производительность для программ с большим количеством долгоживущих объектов.

`collect([generation])`

Запускает полную сборку мусора. Эта функция проверяет все поколения и возвращает количество обнаруженных недоступных объектов. В необязательном аргументе *generation* передается целое число в диапазоне 0–2, которое определяет номер поколения, в котором будет выполняться сборка мусора.

`disable()`

Отключает механизм сборки мусора.

`enable()`

Включает механизм сборки мусора.

`garbage`

Переменная, содержащая доступный только для чтения список экземпляров пользовательских классов, которые больше не используются, но которые не могут быть удалены, потому что они связаны циклическими ссылками и имеют метод `__del__()`. Такие объекты не могут быть удалены сборщиком мусора, так как, чтобы разорвать циклическую связь, интерпретатор должен сначала произвольно удалить один из объектов. При этом не существует надежного способа определить, должен ли метод `__del__()` остающегося объекта, вовлеченного в циклическую ссылку, выполнить какие-то важные операции над объектом, который только что был удален.

`get_count()`

Возвращает кортеж (*count0*, *count1*, *count2*) с текущим количеством объектов в каждом поколении.

`get_debug()`

Возвращает текущие состояния флагов отладки.

`get_objects()`

Возвращает список всех объектов, находящихся под контролем сборщика мусора. За исключением возвращаемого списка.

`get_referrers(obj1, obj2, ...)`

Возвращает список всех объектов, которые непосредственно ссылаются на объекты *obj1*, *obj2* и так далее. Возвращаемый список может содержать объекты, которые еще не были утилизированы сборщиком мусора, а также объекты, находящиеся в стадии создания.

`get_referents(obj1, obj2, ...)`

Возвращает список объектов, на которые ссылаются объекты *obj1*, *obj2* и так далее. Например, если *obj1* является контейнером, для него будет возвращен список объектов, содержащихся в нем.

`get_threshold()`

Возвращает кортеж с текущими пороговыми значениями запуска сборщика мусора.

`isenabled()`

Возвращает `True`, если механизм сборки мусора включен.

`set_debug(`*`flags`*`)`

Устанавливает **отладочные флаги сборщика мусора, которые могут использоваться** для отладки поведения сборщика мусора. В аргументе *`flags`* передается целое число, составленное с помощью битовой операции ИЛИ из констант `DEBUG_STATS`, `DEBUG_COLLECTABLE`, `DEBUG_UNCOLLECTABLE`, `DEBUG_INSTANCES`, `DEBUG_OBJECTS`, `DEBUG_SAVEALL` и `DEBUG_LEAK`. Флаг `DEBUG_LEAK` является, пожалуй, наиболее полезным, потому что он вынуждает механизм сборщика мусора выводить информацию, которая может пригодиться при отладке программ с утечками памяти.

`set_threshold(`*`threshold0`* `[, ` *`threshold1`*`[, ` *`threshold2`*`]])`

Устанавливает частоту запуска сборщика мусора. Объекты распределяются по трем поколениям, где поколение 0 содержит самые молодые объекты, а поколение 2 – самые старые. Объекты, пережившие этап сборки мусора, перемещаются в следующее, более старшее поколение. Достигнув **поколения** 2, объект остается в нем. В аргументе *`threshold0`* передается число, разность между количеством операций создания новых объектов и количеством операций удаления объектов, по достижении которого запускается сборка мусора среди объектов поколения 0. В аргументе *`threshold1`* передается число, определяющее количество запусков процедуры сборки мусора среди объектов поколения 0, по достижении которого запускается сборка мусора среди объектов поколения 1. В аргументе *`threshold2`* передается число, определяющее количество запусков процедуры сборки мусора среди объектов поколения 1, по достижении которого запускается сборка мусора среди объектов поколения 2. По умолчанию используются пороговые значения `(700,10,10)`. Если в аргументе *`threshold0`* передать значение `0`, сборка мусора будет отключена.

Примечания

- Объекты, связанные циклическими ссылками и имеющие метод `__del__()`, не утилизируются сборщиком мусора и помещаются в список `gc.garbage` (объекты, которые не могут быть удалены). Эти объекты не удаляются из-за сложностей, связанных с необходимостью выполнения заключительных операций этими объектами.
- Функции `get_referrers()` и `get_referents()` применяются только к объектам, поддерживающим сборку мусора. Кроме того, эти функции предназначены только для отладки. Они не должны использоваться в каких-либо других целях.

Модуль inspect

Модуль `inspect` используется для сбора информации о существующих объектах, такой как имена и значения атрибутов, строки документирования, исходный программный код, кадры стеков и так далее.

`cleandoc(doc)`

Приводит в порядок строку документирования *doc*, замещая все символы табуляции пробелами и удаляя отступы, которые могли быть добавлены, чтобы выровнять строку документирования в соответствии с другими инструкциями в функции или в методе.

`currentframe()`

Возвращает объект кадра стека, соответствующий кадру стека вызывающей функции.

`formatargspec(args [, varags [, varkw [, defaults]]])`

Возвращает отформатированную строку, представляющую значения, возвращаемые функцией `getargspec()`.

`formatargvalues(args [, varargs [, varkw [, locals]]])`

Возвращает отформатированную строку, представляющую значения, возвращаемые функцией `getargvalues()`.

`getargspec(func)`

Для заданной функции *func* возвращает именованный кортеж `ArgSpec(args, varargs, varkw, defaults)`, где *args* – это список имен аргументов функции *func*, *varargs* – имя аргумента, начинающегося с символа `*` (если имеется), *varkw* – имя аргумента, начинающегося с символов `**` (если имеется), и *defaults* – кортеж значений по умолчанию для аргументов или `None`, если аргументы со значениями по умолчанию отсутствуют. Если в функции *func* имеются аргументы со значениями по умолчанию, кортеж *defaults* представляет значения последних *n* аргументов в списке *args*, где *n* – результат вызова функции `len(defaults)`.

`getargvalues(frame)`

Возвращает значения аргументов, переданных функции с кадром стека *frame*. Возвращает кортеж `ArgInfo(args, varargs, varkw, locals)`, где *args* – список имен аргументов, *varargs* – имя аргумента, начинающегося с символа `*` (если имеется), *varkw* – имя аргумента, начинающегося с символов `**` (если имеется), и *locals* – локальный словарь кадра стека.

`getclasstree(classes [, unique])`

Получая список *classes* связанных классов, эта функция воссоздает иерархию их наследования. Иерархия представлена как коллекция вложенных списков, где каждый элемент списка является списком классов, наследующих класс, непосредственно предшествующий этому списку. Каждый элемент вложенного списка является кортежем из 2 элементов (*cls*, *bases*), где *cls* – объект класса, а *bases* – кортеж базовых классов. Если в аргументе *unique* передается значение `True`, каждый класс будет включаться в возвращаемый список только один раз. В противном случае, когда используется множественное наследование, класс может появляться в списке несколько раз.

`getcomments(object)`

Возвращает строку, содержащую комментарии, непосредственно предшествующие объявлению объекта *object* в исходном программном коде на

языке Python. Если объект является модулем, возвращаются комментарии, расположенные в начале модуля. В случае отсутствия комментариев возвращается None.

`getdoc(object)`

Возвращает строку документирования объекта *object*. Перед этим строка документирования обрабатывается функцией `cleandoc()`.

`getfile(object)`

Возвращает имя файла, в котором находится определение объекта *object*. Может вернуть `TypeError`, если эта информация не имеет смысла или недоступна (например, для встроенных функций).

`getframeinfo(frame [, context])`

Возвращает именованный кортеж `Traceback(filename, lineno, function, code_context, index)`, содержащий сведения об объекте *frame* с информацией о кадре стека. Поля *filename* и *lineno* определяют местоположение в исходном программном коде. Аргумент *context* определяет количество строк контекста, которые будут извлекаться из исходного программного кода. Поле *code_context* в возвращаемом кортеже содержит список строк, составляющих контекст. Поле *index* – числовой индекс строки в этом списке, соответствующей кадру стека.

`getinnerframes(traceback [, context])`

Возвращает список записей в кадре стека для кадра объекта *traceback* с трассировочной информацией и для всех вложенных кадров. Каждая запись является кортежем из 6 элементов (*frame, filename, lineno, funcname,* `code_`*context, index*). Поля *filename*, *lineno*, *context*, `code_`*context* и *index* имеют тот же смысл, что и в кортеже, возвращаемом функцией `getframeinfo()`.

`getmembers(object [, predicate])`

Возвращает все атрибуты объекта *object*. Обычно атрибуты выбираются с помощью атрибута `__dict__` объекта, но эта функция может возвращать атрибуты, хранящиеся в других местах (например, строку документирования – из атрибута `__doc__`, имя объекта – из атрибута `__name__` и так далее). Атрибуты возвращаются в виде списка пар (*name, value*). В необязательном аргументе *predicate* можно передать функцию, которая принимает атрибут объекта в качестве аргумента и возвращает `True` или `False`. В этом случае в список будут включены только те атрибуты, для которых функция *predicate* вернет `True`. В качестве функции *predicate* можно использовать такие функции, как `isfunction()` и `isclass()`.

`getmodule(object)`

Возвращает модуль, в котором был объявлен объект *object* (если это возможно).

`getmoduleinfo(path)`

Возвращает информацию о том, как Python интерпретирует путь к файлу *path*. Если *path* не является модулем Python, возвращается `None`. В противном случае возвращается именованный кортеж `ModuleInfo(name, suffix, mode, module_type)`, где *name* – это имя модуля, *suffix* – расширение имени

файла, *mode* – режим открытия файла модуля и *module_type* – целочисленный код, определяющий тип модуля. Коды типов модулей определены в модуле `imp` как:

Тип модуля	Описание
`imp.PY_SOURCE`	Файл с исходным программным кодом на языке Python
`imp.PY_COMPILED`	Скомпилированный файл с байт-кодом
`imp.C_EXTENSION`	Динамически загружаемое расширение, написанное на языке C
`imp.PKG_DIRECTORY`	Каталог пакета
`imp.C_BUILTIN`	Встроенный модуль
`imp.PY_FROZEN`	Фиксированный модуль

`getmodulename(`*`path`*`)`

Возвращает имя модуля, заданного путем *path*. Если аргумент *path* не выглядит как модуль Python, возвращается `None`.

`getmro(`*`cls`*`)`

Возвращает кортеж классов, который является представлением порядка поиска методов в классе *cls*. Дополнительные подробности приводятся в главе 7 «Классы и объектно-ориентированное программирование».

`getouterframes(`*`frame`*` [, `*`context`*`])`

Возвращает список записей из текущего кадра стека и всех объемлющих кадров. Этот список является представлением последовательности вызовов функций, где первый элемент содержит информацию о кадре стека. Каждая запись является кортежем из 6 элементов (*frame*, *filename*, *lineno*, *funcname*, *code_context*, *index*), поля которого имеют тот же смысл, что и в кортеже, возвращаемом функцией `getinnerframes()`. Аргумент *context* имеет то же назначение, что и в функции `getframeinfo()`.

`getsourcefile(`*`object`*`)`

Возвращает имя файла с исходным программным кодом на языке Python, где находится определение объекта *object*.

`getsourcelines(`*`object`*`)`

Возвращает кортеж (*sourcelines*, *firstline*), соответствующий определению объекта *object*. Поле *sourcelines* – список строк с исходным программным кодом, а поле *firstline* – номер первой строки с исходным программным кодом. Если исходный код невозможно получить, возбуждает исключение `IOError`.

`getsource(`*`object`*`)`

Возвращает исходный программный код объекта *object* в виде единственной строки. Если исходный код невозможно получить, возбуждает исключение `IOError`.

`isabstract(`*`object`*`)`

Возвращает `True`, если объект *`object`* является абстрактным базовым классом.

`isbuiltin(`*`object`*`)`

Возвращает `True`, если объект *`object`* является встроенной функцией.

`isclass(`*`object`*`)`

Возвращает `True`, если объект *`object`* является классом.

`iscode(`*`object`*`)`

Возвращает `True`, если объект *`object`* является объектом с программным кодом.

`isdatadescriptor(`*`object`*`)`

Возвращает `True`, если объект *`object`* является дескриптором данных. Таковыми считаются объекты, определяющие оба метода: `__get__()` и `__set__()`.

`isframe(`*`object`*`)`

Возвращает `True`, если объект *`object`* является кадром стека.

`isfunction(`*`object`*`)`

Возвращает `True`, если объект *`object`* является функцией.

`isgenerator(`*`object`*`)`

Возвращает `True`, если объект *`object`* является генератором.

`isgeneratorfunction(`*`object`*`)`

Возвращает `True`, если объект *`object`* является функцией-генератором. Отличие от функции `isgenerator()` состоит в том, что данная функция проверяет, является ли объект функцией, которая при вызове создает генератор. Не может использоваться для проверки, является ли генератор активным.

`ismethod(`*`object`*`)`

Возвращает `True`, если объект *`object`* является методом.

`ismethoddescriptor(`*`object`*`)`

Возвращает `True`, если объект *`object`* является дескриптором метода. Таковыми считаются объекты, не являющиеся методами класса и определяющие метод `__get__()`, но не определяющие метод `__set__()`.

`ismodule(`*`object`*`)`

Возвращает `True`, если объект *`object`* является модулем.

`isroutine(`*`object`*`)`

Возвращает `True`, если объект *`object`* является пользовательской или встроенной функцией или методом.

`istraceback(`*`object`*`)`

Возвращает `True`, если объект *`object`* является объектом с трассировочной информацией.

`stack([context])`

Возвращает список записей, соответствующих текущему кадру стека. Каждая запись является кортежем из 6 элементов (*frame*, *filename*, *lineno*, *funcname*, *code_context*, *index*), содержащих ту же информацию, которая возвращается функцией `getinnerframes()`. Аргумент *context* определяет количество строк контекста, которые должны возвращаться для каждой записи кадра.

`trace([context])`

Возвращает список записей для кадров стека между текущим кадром и кадром, в котором возникло исключение. Первая запись соответствует текущей функции, а последняя – кадру, в котором возникло исключение. Аргумент *context* определяет количество строк контекста, которые должны возвращаться для каждой записи кадра.

Модуль marshal

Модуль `marshal` используется для сериализации объектов в «недокументированные» форматы представления данных, используемые в языке Python. Модуль `marshal` похож на модули `pickle` и `shelve`, но обладает меньшими возможностями и может использоваться только при работе с простыми объектами. Он вообще не должен использоваться для реализации сохранения объектов (для этого лучше использовать модуль `pickle`). Однако при работе с простыми встроенными типами модуль `marshal` обеспечивает очень быстрый способ сохранения и загрузки данных.

`dump(value, file [, version])`

Сохраняет объект *value* в объекте *file* открытого файла. Если объект *value* относится к неподдерживаемым типам, возбуждается исключение `ValueError`. Аргумент *version* – целое число, определяющее используемый формат данных. Код формата по умолчанию определяется переменной `marshal.version` и в настоящее время имеет значение 2. Версия 0 – это устаревший формат, использовавшийся более ранними версиями Python.

`dumps(value [,version])`

Возвращает строку, записанную функцией `dump()`. Если объект *value* относится к неподдерживаемым типам, возбуждается исключение `ValueError`. Назначение аргумента *version* описано выше.

`load(file)`

Читает и возвращает следующее значение из объекта *file* открытого файла. Если функции не удалось прочитать допустимое значение, будет возбуждено исключение `EOFError`, `ValueError` или `TypeError`. Формат данных в файле определяется автоматически.

`loads(string)`

Читает и возвращает следующее значение из строки *string*.

Примечания

- Данные сохраняются в платформонезависимом формате.
- Поддерживаются только `None`, целые числа, длинные целые числа, числа с плавающей точкой, комплексные числа, строки, строки Юникода, кортежи, списки, словари и объекты с программным кодом. Списки, кортежи и словари могут содержать только объекты поддерживаемых типов. Экземпляры классов и рекурсивные ссылки в списках, кортежах и словарях не поддерживаются.
- Целые числа могут быть преобразованы в длинные целые, если встроенный тип `int` не обладает достаточной точностью для их представления, например если сериализованное целое число содержит 64 бита, а чтение выполняется на платформе, поддерживающей 32-битные целые числа.
- Модуль `marshal` не обеспечивает защиту от ошибочных или злонамеренных данных и не должен использоваться для чтения данных, полученных из непроверенных источников.
- Модуль `marshal` имеет значительно более высокую производительность, чем модуль `pickle`, и имеет менее широкие возможности.

Модуль pickle

Модуль `pickle` используется для преобразования объектов Python в последовательность байтов, пригодную для сохранения в файле, в базе данных или для передачи по сети. Этот процесс называют по-разному: *сериализацией* или *маршаллингом*. Получающаяся последовательность байтов может быть снова преобразована в серию объектов Python в процессе восстановления.

Ниже перечислены функции, которые могут использоваться для преобразования объектов в последовательности байтов.

`dump(`*`object`*`, `*`file`*` [, `*`protocol`*` ])`

Выводит сериализованное представление объекта *`object`* в объект файла *`file`*. Аргумент *`protocol`* определяет формат выводимых данных. Протокол 0 (по умолчанию) – это текстовый формат, обратно совместимый с более ранними версиями Python. Протокол 1 – это двоичный формат, который также совместим с большинством предыдущих версий Python. Протокол 2 – это самый новый формат, который обеспечивает наиболее эффективный способ сохранения классов и экземпляров. Протокол 3 используется в Python 3 и несовместим с предыдущими версиями интерпретатора. Если в аргументе *`protocol`* передается отрицательное значение, выбирается наиболее современный протокол. Самая последняя версия протокола хранится в переменной `pickle.HIGHEST_PROTOCOL`. Если объект не поддерживает возможность сериализации, возбуждается исключение `pickle.PicklingError`.

`dumps(`*`object`*` [, `*`protocol`*`])`

То же, что и функция `dump()`, но возвращает строку, содержащую сериализованные данные.

Следующий пример демонстрирует, как можно было бы использовать эти функции для сохранения объектов в файле:

```
f = open('myfile', 'wb')
pickle.dump(x, f)
pickle.dump(y, f)
... сохранить еще объекты ...
f.close()
```

Следующие функции могут использоваться для восстановления сериализованных объектов.

`load(`*`file`*`)`

Загружает сериализованное представление объекта из объекта *file* файла и возвращает готовый объект. Протокол указывать не требуется, так как он определяется автоматически. Если файл содержит поврежденные данные, которые не могут быть декодированы, функция возбуждает исключение `pickle.UnpicklingError`. По достижении конца файла возбуждается исключение `EOFError`.

`loads(`*`string`*`)`

То же, что и функция `load()`, но сериализованное представление объекта из строки.

Следующий пример демонстрирует, как можно было бы использовать эти функции для восстановления объектов:

```
f = open('myfile', 'rb')
x = pickle.load(f)
y = pickle.load(f)
... загрузить еще объекты ...
f.close()
```

При выполнении загрузки не требуется указывать протокол или какую-нибудь другую информацию о типе загружаемого объекта. Эта информация сохраняется вместе с сериализованными данными.

Если речь идет о сохранении более чем одного объекта, можно просто выполнить серию вызовов функции `dump()` и `load()`, как было показано в предыдущих примерах. При выполнении серии вызовов функций достаточно просто гарантировать, что последовательность вызовов функции `load()` соответствует последовательности вызовов `dump()`, использовавшихся для записи объектов в файл.

Когда дело доходит до сложных структур данных с циклическими или разделяемыми ссылками, использование функций `dump()` и `load()` может оказаться проблематичным, потому что они не поддерживают возможность сохранения информации об объектах, которые уже были сохранены или восстановлены. Отсутствие такой возможности может привести к чрезмерному увеличению размеров файлов и вызывать ошибки при восстановлении взаимоотношений между загруженными объектами. Альтернативный подход заключается в использовании объектов классов `Pickler` и `Unpickler`.

`Pickler(`*`file`* `[, `*`protocol`* `])`

Создает объект, который записывает данные в объект *file* файла, используя указанный протокол *protocol* сериализации. Экземпляр *p* класса `Pickler` имеет метод *p*`.dump(x)`, который записывает объект *x* в объект *file* файла. При сохранении объекта *x* запоминается его идентичность.[1] Если впоследствии повторно будет выполнена попытка сохранить объект *x* с помощью метода *p*`.dump()`, вместо новой копии в файл будет записана ссылка на объект, сохраненный ранее. Метод *p*`.clear_memo()` очищает внутренний словарь, используемый для сохранения информации о ранее записанных объектах. Эту возможность можно было бы использовать, чтобы записать свежую копию сохраненного ранее объекта (например, если изменилось его значение с момента последнего вызова функции `dump()`).

`Unpickler(`*`file`*`)`

Создает объект, который читает данные из объекта *file* файла. Экземпляр *u* класса `Unpickler` имеет метод *u*`.load()`, который загружает данные из файла и возвращает новый объект. Объект `Unpickler` запоминает объекты, которые он возвращал, потому что исходный файл может содержать ссылки на объекты, сохраненные объектом `Pickler`. В этом случае метод *u*`.load()` возвращает ссылку на ранее загруженный объект.

Модуль `pickle` может работать с большинством разновидностей обычных объектов Python, включая:

- `None`
- Числа и строки
- Кортежи, списки и словари, содержащие объекты только тех типов, которые поддерживаются модулем `pickle`
- Экземпляры пользовательских классов, объявленных в модуле на верхнем уровне

Когда сохраняются экземпляры пользовательских классов, данные об экземпляре – единственная информация, которая сохраняется. Соответствующее определение класса не сохраняется, вместо этого сохраненные данные просто содержат *имя* соответствующего класса и имя модуля, где этот класс объявлен. Когда происходит восстановление такого экземпляра, автоматически выполняется импортирование модуля, в котором находится объявление класса, – для получения доступа к определению класса при воссоздании экземпляра. Следует также отметить, что при восстановлении экземпляра метод `__init__()` класса не вызывается. Воссоздание и восстановление данных в экземпляре производится другими способами.

Одно из ограничений, накладываемых на экземпляры классов, заключается в том, что соответствующие определения классов должны находиться на верхнем уровне модуля (то есть это не могут быть вложенные классы). Кроме того, если определение класса экземпляра первоначально было выполнено в модуле `__main__`, оно должно быть загружено вручную, до попытки восстановления сохраненного объекта (поскольку интерпретатор не знает,

[1] Числовой идентификатор. – *Прим. перев.*

как автоматически загрузить определение требуемого класса обратно в модуль `__main__` в процессе восстановления).

Обычно не требуется делать что-то особенное с пользовательскими классами, чтобы обеспечить их совместимость с модулем `pickle`. Однако класс может предоставлять собственные версии методов сохранения и восстановления информации о своем состоянии, реализовав специальные методы `__getstate__()` и `__setstate__()`. Метод `__getstate__()` должен возвращать объект, пригодный для сохранения (такой как строка или кортеж), представляющий состояние оригинального объекта. Метод `__setstate__()` должен принимать сохраненный объект и на его основе восстанавливать оригинальный объект. Если эти методы не определены, по умолчанию сохраняется атрибут `__dict__` экземпляра. Следует отметить, что если эти методы определены, они также будут использоваться модулем `copy` для операций поверхностного и глубокого копирования.

Примечания

- В Python 2 имеется модуль `cPickle`, содержащий реализацию функций модуля `pickle` на языке C. Он обладает гораздо более высокой производительностью по сравнению с модулем `pickle`, но ограничивает возможность создания производных классов от `Pickler` и `Unpickler`. В Python 3 также имеется модуль, содержащий реализацию на языке C, но он не используется непосредственно (модуль `pickle` использует его автоматически).
- Формат данных, используемый модулем `pickle`, характерен для языка Python и не должен рассматриваться как совместимый с какими-либо внешними стандартами, такими как XML.
- Везде, где только возможно, вместо модуля `marshal` должен использоваться модуль `pickle`, потому что модуль `pickle` является более гибким, используемый им формат кодирования данных описан в документации и в нем выполняется проверка на наличие ошибок.
- Из-за проблем, связанных с безопасностью, программы не должны восстанавливать данные из файлов, полученных из непроверенных источников.
- Использование модуля `pickle` с типами, объявленными в модулях расширений, является намного более сложной задачей, чем было описано здесь. Разработчики расширений обязательно должны ознакомиться с подробным описанием низкоуровневого протокола в электронной документации, чтобы обеспечить возможность сохранения объектов с помощью модуля `pickle`. В частности, с особенностями реализации специальных методов `__reduce__()` и `__reduce_ex__()`, которые используются модулем `pickle` для создания последовательности байтов.

Модуль sys

Модуль `sys` содержит переменные и функции, имеющие отношение к интерпретатору и его окружению.

Переменные

В модуле объявлены перечисленные далее переменные.

`api_version`

Целочисленное представление версии **C API интерпретатора Python. Используется** при работе с модулями расширений.

`argv`

Список параметров командной строки, передаваемых программе. Элемент `argv[0]` хранит имя программы.

`builtin_module_names`

Кортеж с именами модулей, встроенных в исполняемый файл интерпретатора Python.

`byteorder`

Порядок следования байтов, используемый аппаратной платформой: `'little'` — обратный порядок следования байтов, `'big'` — прямой.

`copyright`

Строка с текстом, содержащим упоминание об авторских правах.

`__displayhook__`

Оригинальное значение функции `displayhook()`.

`dont_write_bytecode`

Логический флаг, который определяет, должен ли интерпретатор Python создавать файлы с байт-кодом (с расширением `.pyc` или `.pyo`) при импортировании модулей. Начальное значение `True`, если интерпретатор не был вызван с ключом `-B`. Программа может изменять значение этой переменной по своему усмотрению.

`dllhandle`

Целочисленный идентификатор для Python DLL (используется в Windows).

`__excepthook__`

Оригинальное значение функции `excepthook()`.

`exec_prefix`

Каталог, куда были установлены платформозависимые файлы Python.

`executable`

Строка, содержащая имя выполняемого файла интерпретатора.

`flags`

Объект, представляющий различные параметры командной строки, которые были переданы при запуске самому интерпретатору Python. Ниже приводится список атрибутов объекта `flags` вместе с соответствующими параметрами командной строки, включающими флаги. Эти атрибуты доступны только для чтения.

Атрибут	Параметр командной строки
flags.debug	-d
flags.py3k_warning	-3
flags.division_warning	-Q
flags.division_new	-Qnew
flags.inspect	-i
flags.interactive	-i
flags.optimize	-O или -OO
flags.dont_write_bytecode	-B
flags.no_site	-S
flags.ignore_environment	-E
flags.tabcheck	-t или -tt
flags.verbose	-v
flags.unicode	-U

`float_info`

Объект, который хранит информацию о внутреннем представлении чисел с плавающей точкой. Значения атрибутов этого объекта взяты из заголовочного файла `float.h` языка C.

Атрибут	Описание
float_info.epsilon	Разность между 1.0 и ближайшим числом с плавающей точкой больше 1.0.
float_info.dig	Количество десятичных знаков, которые могут быть представлены без изменений после округления.
float_info.mant_dig	Количество цифр мантиссы в системе счисления по основанию, указанному в атрибуте float_info.radix.
float_info.max	Максимально возможное число с плавающей точкой.
float_info.max_exp	Максимальная величина экспоненты в системе счисления по основанию, указанному в атрибуте float_info.radix.
float_info.max_10_exp	Максимальная величина экспоненты в системе счисления по основанию 10.
float_info.min	Минимально возможное положительное число с плавающей точкой.
float_info.min_exp	Минимальная величина экспоненты в системе счисления по основанию, указанному в атрибуте float_info.radix.

Атрибут	Описание
float_info.min_10_exp	Минимальная величина экспоненты в системе счисления по основанию 10.
float_info.radix	Основание системы счисления для показателя степени.
float_info.rounds	Алгоритм округления (-1 – не определено, 0 – в сторону нуля, 1 – до ближайшего значения, 2 – в сторону положительной бесконечности, 3 – в сторону отрицательной бесконечности).

`hexversion`

Целое число, в шестнадцатеричном представлении которого закодирована информация о номере версии, содержащемся в переменной `sys.version_info`. Значение этой переменной всегда гарантированно увеличивается с выходом новой версии интерпретатора.

`last_type, last_value, last_traceback`

Значения этих переменных устанавливаются, когда появляется необработанное исключение и интерпретатор выводит сообщение об ошибке. В переменной `last_type` сохраняется тип последнего исключения, в переменной `last_value` – экземпляр последнего исключения, а в переменной `last_traceback` – объект с трассировочной информацией. Обратите внимание, что в многопоточных приложениях не гарантируется достоверность информации в этих переменных, поэтому вместо них рекомендуется пользоваться функцией `sys.exc_info()`.

`maxint`

Максимально возможное значение целого числа (только в Python 2).

`maxsize`

Максимально возможное целое число, поддерживаемое типом `size_t` языка C в системе. Это значение определяет максимально возможную длину строк, списков, словарей и других встроенных типов.

`maxunicode`

Целое число, определяющее наибольший кодовый пункт Юникода, который может быть представлен. По умолчанию имеет значение 65 535 для 16-битной кодировки UCS-2. Если при сборке интерпретатор Python был настроен на использование кодировки UCS-4, это число будет больше.

`modules`

Словарь, который отображает имена модулей в объекты модулей.

`path`

Список строк, определяющих путь поиска модулей. Первый элемент списка всегда содержит путь к каталогу, в котором находился сценарий, использованный для запуска интерпретатора (если доступен). Подробности приводятся в главе 8 «Итераторы и генераторы».

`platform`

Строка, идентифицирующая платформу, например `'linux-i386'`.

`prefix`

Каталог, куда были установлены платформонезависимые файлы Python.

`ps1, ps2`

Строки, содержащие текст основного и дополнительного приглашения к вводу интерпретатора. Изначально переменная `ps1` имеет значение `'>>> '`, а `ps2` – значение `'... '`. При назначении этим переменным других значений для получения текста приглашения будут использоваться методы `str()` назначенных объектов.

`py3kwarning`

В Python 2 этот флаг устанавливается в значение `True`, когда интерпретатор запускается с ключом `-3`.

`stdin, stdout, stderr`

Объекты файлов, соответствующие потокам стандартного ввода, стандартного вывода и стандартного вывода сообщений об ошибках. Переменная `stdin` используется функциями `raw_input()` и `input()`. Переменная `stdout` используется инструкцией `print` для вывода значений аргументов и функциями `raw_input()` и `input()` – для вывода приглашения к вводу. Переменная `stderr` используется интерпретатором для вывода приглашения к вводу и сообщений об ошибках. Этим переменным могут быть назначены любые объекты, поддерживающие метод `write()`, принимающий единственный строковый аргумент.

`__stdin__, __stdout__, __stderr__`

Переменные, содержащие значения `stdin`, `stdout` и `stderr`, полученные в момент запуска интерпретатора.

`tracebacklimit`

Максимальное количество уровней трассировочной информации, которая выводится при появлении необработанного исключения. Значение по умолчанию **1000. Значение 0 подавляет вывод трассировочной информации**, при этом выводятся только тип исключения и информация из него.

`version`

Строка с номером версии.

`version_info`

Информация о версии в виде кортежа (*major, minor, micro, releaselevel, serial*). Все значения являются целочисленными, за исключением *releaselevel*, которое является строкой `'alpha'`, `'beta'`, `'candidate'` или `'final'`.

`warnoptions`

Список аргументов параметра командной строки `-W`, полученного интерпретатором.

`winver`

Номер версии, который обычно формируется из ключей реестра в системе Windows.

Функции

Ниже приводятся функции, доступные в модуле `sys`:

`_clear_type_cache()`

Очищает внутренний кэш типов. Чтобы оптимизировать поиск наиболее часто используемых методов, внутри интерпретатора имеется небольшой кэш на 1024 записи. Этот кэш позволяет ускорить повторные попытки поиска, особенно в программном коде, где используются классы с глубокой иерархией наследования. Обычно этот кэш очищать не требуется, но эта функция может пригодиться при решении некоторых проблем освобождения памяти, связанных с подсчетом ссылок. Например, когда метод в кэше удерживает ссылку на объект, который требуется удалить.

`_current_frames()`

Возвращает словарь, отображающий идентификаторы потоков выполнения на самые верхние кадры стека, для потоков, которые были активны в момент вызова функции. Эта информация может пригодиться при разработке инструментов отладки многопоточных приложений (то есть для поиска причин взаимоблокировок). Имейте в виду, что значения, возвращаемые функцией, представляют собой всего лишь «мгновенный снимок» интерпретатора в момент вызова функции. К тому времени, когда функция вернет результаты, потоки могут уже выполняться в другом месте.

`displayhook([`*`value`*`])`

Эта функция вызывается для вывода результатов выражений, когда интерпретатор выполняется в интерактивном режиме. По умолчанию она выводит значение `repr(`*`value`*`)` в поток стандартного вывода и сохраняет его в переменной `__builtin__._`. Имеется возможность переопределять значение `displayhook`, чтобы при необходимости обеспечить иное поведение.

`excepthook(`*`type`*`,`*`value`*`,`*`traceback`*`)`

Эта функция вызывается при появлении необработанного исключения. В аргументе *`type`* ей передается класс исключения, в аргументе *`value`* – значение, переданное инструкции `raise`, а в аргументе *`traceback`* – объект с трассировочной информацией. По умолчанию она выводит в поток стандартного вывода сообщений об ошибках информацию об исключении и трассировочную информацию. Однако имеется возможность переопределить эту функцию и реализовать альтернативный способ реакции на необработанные исключения (что может оказаться полезным в специализированных приложениях, таких как отладчики или сценарии CGI).

`exc_clear()`

Очищает всю информацию, связанную с последним исключением. При этом очищается только информация, имеющая отношение к вызывающему потоку выполнения.

`exc_info()`

Возвращает кортеж (*type*, *value*, *traceback*) с информацией об исключении, обрабатываемом в текущий момент. В аргументе *type* передается тип исключения, в аргументе *value* – параметры инструкции `raise`, а в аргументе *traceback* – объект с трассировочной информацией о той точке в стеке вызовов, где возникло исключение. Если в текущий момент времени никакое исключение не обрабатывается, возвращает `None`.

`exit([n])`

Завершает работу интерпретатора, возбуждая исключения `SystemExit`. В аргументе *n* передается целое число – код завершения. Значение 0 рассматривается как признак нормального завершения (по умолчанию); ненулевые значения интерпретируются как признак ошибки. Если в аргументе *n* передать значение, не являющееся целым числом, оно будет выведено в поток `sys.stderr`, а завершение работы будет произведено с кодом 1.

`getcheckinterval()`

Возвращает величину интервала проверки, который определяет, как часто интерпретатор будет проверять наличие сигналов, необходимость переключения потоков выполнения и других периодических событий.

`getdefaultencoding()`

Возвращает кодировку, используемую по умолчанию для преобразований строк Юникода. Возвращаемое значение является строкой, такой как `'ascii'` или `'utf-8'`. Установка кодировки по умолчанию производится модулем `site`.

`getdlopenflags()`

Возвращает флаги, которые передаются функции `dlopen()`, написанной на языке C, во время загрузки модулей расширений в UNIX. Дополнительные подробности смотрите в документации к модулю `dl`.

`getfilesystemencoding()`

Возвращает кодировку символов, используемую операционной системой для отображения символов Юникода в именах файлов. В Windows возвращает `'mbcs'`, а в Macintosh OS X – `'utf-8'`. В UNIX кодировка зависит от региональных настроек, и функция будет возвращать значение параметра настройки `CODESET`. Может возвращать `None`, если в системе используется кодировка по умолчанию.

`_getframe([depth])`

Возвращает объект кадра стека вызовов. Если аргумент *depth* опущен или равен нулю, возвращается самый верхний кадр. В противном случае возвращается кадр, расположенный ниже текущего кадра стека на указанное число уровней. Например, вызов `_getframe(1)` вернет кадр стека вызывающей функции. При передаче в аргументе *depth* недопустимого значения возбуждает исключение `ValueError`.

`getprofile()`

Возвращает функцию профилирования, установленную вызовом функции `setprofile()`.

`getrecursionlimit()`

Возвращает ограничение на количество рекурсивных вызовов функций.

`getrefcount(`*`object`*`)`

Возвращает значение счетчика ссылок на объект *object*.

`getsizeof(`*`object`* `[, `*`default`*`])`

Возвращает размер объекта *object* в байтах. Вычисления выполняются с помощью специального метода `__sizeof__()` указанного объекта. Если этот метод не определен, возбуждается исключение `TypeError`, если не было указано значение по умолчанию в аргументе *default*. Поскольку на реализацию методов `__sizeof__()` в объектах не накладывается никаких ограничений, то нет никакой гарантии, что возвращаемое значение функции будет точно соответствовать объему занимаемой памяти. Однако для встроенных типов, таких как списки или строки, функция возвращает корректное значение.

`gettrace()`

Возвращает функцию трассировки, установленную функцией `settrace()`.

`getwindowsversion()`

Возвращает кортеж (*major*,*minor*,*build*,*platform*,*text*), описывающий версию используемой системы Windows. Поле *major* содержит основной номер версии. Например, значение 4 соответствует операционной системе Windows NT 4.0, а значение 5 – Windows 2000 и Windows XP. Поле *minor* содержит младший номер версии. Например, значение 0 соответствует операционной системе Windows 2000,[1] а значение 1 – Windows XP. Поле *build* содержит номер сборки. Поле *platform* – целое число с информацией о платформе и может принимать одно из типичных значений: 0 (Win32s в Windows 3.1), 1 (Windows 95, 98 или Me), 2 (Windows NT, 2000, XP) или 3 (Windows CE). Поле *text* содержит строку с дополнительной информацией, такой как `"Service Pack 3"`.

`setcheckinterval(`*`n`*`)`

Устанавливает количество инструкций виртуальной машины Python, которые должен выполнить интерпретатор, прежде чем он проверит наступление периодического события, такого как сигнал или необходимость переключения контекста потока выполнения. По умолчанию устанавливается значение 10.

`setdefaultencoding(`*`enc`*`)`

Устанавливает кодировку по умолчанию. В аргументе *enc* должна передаваться строка, такая как `'ascii'` или `'utf-8'`. Эта функция определена только в модуле `site`. Может вызываться из пользовательских модулей `sitecustomize`.

`setdlopenflags(`*`flags`*`)`

Устанавливает флаги, которые в дальнейшем будут передаваться функции `dlopen()`, написанной на языке C, которая используется для загрузки моду-

[1] При старшем номере версии 5. – *Прим. перев.*

лей расширений в UNIX. Эти флаги оказывают влияние на способ разрешения имен в библиотеках и в других модулях расширений. В аргументе *flags* передается битная маска, составленная из значений флагов, объединенных с помощью битовой операции ИЛИ, которые определены в модуле dl, например sys.setdlopenflags(dl.RTLD_NOW | dl.RTLD_GLOBAL).

setprofile(*pfunc*)

Устанавливает функцию профилирования, которая может использоваться реализацией профилировщика исходных текстов.

setrecursionlimit(*n*)

Устанавливает ограничение на количество рекурсивных вызовов функций. По умолчанию устанавливается значение 1000. Обратите внимание, что операционная система может накладывать свои ограничения на размер стека, поэтому установка слишком большого значения может вызывать аварийное завершение процесса интерпретатора Python с сообщением «Segmentation Fault» (ошибка сегментации) или «Access Violation» (нарушение прав доступа).

settrace(*tfunc*)

Устанавливает функцию трассировки, которая может использоваться реализацией отладчика. Дополнительная информация об отладчике Python приводится в главе 11.

Модуль traceback

Модуль traceback используется для сбора и вывода трассировочной информации о программе после появления исключения. Функции в этом модуле оперируют объектами с трассировочной информацией, такими как в третьем элементе возвращаемого значения функции sys.exc_info(). В основном этот модуль может использоваться для реализации нестандартного способа вывода сообщений об ошибках, например когда программы на языке Python выполняются глубоко в недрах сетевого сервера и необходимо организовать вывод трассировочной информации в файл журнала.

print_tb(*traceback* [, *limit* [, *file*]])

Выводит в файл *file* до *limit* записей из объекта *traceback* с трассировочной информацией. Если аргумент *limit* опущен, выводятся все имеющиеся записи. Если аргумент *file* опущен, вывод отправляется в sys.stderr.

print_exception(*type*, *value*, *traceback* [, *limit* [, *file*]])

Выводит в файл *file* информацию об исключении и трассировочную информацию. В аргументе *type* передается тип исключения, а в аргументе *value* – значение исключения. Аргументы *limit* и *file* имеют тот же смысл, что и в функции print_tb().

print_exc([*limit* [, *file*]])

То же, что и print_exception(), но применяется к информации, возвращаемой функцией sys.exc_info().

`format_exc([limit [, file]])`

Возвращает строку, содержащую ту же информацию, которую выводит функция `print_exc()`.

`print_last([limit [, file]])`

То же, что и `print_exception(sys.last_type, sys.last_value, sys.last_traceback, limit, file)`.

`print_stack([frame [, limit [, file]]])`

Выводит трассировочную информацию для точки, откуда был сделан вызов функции. В необязательном аргументе *frame* передается начальный кадр стека. Аргументы *limit* и *file* имеют тот же смысл, что и в функции `print_tb()`.

`extract_tb(traceback [, limit])`

Извлекает ту же трассировочную информацию, что и функция `print_tb()`. Возвращает список кортежей вида (*filename*, *line*, *funcname*, *text*), содержащих информацию, которая обычно выводится в сообщениях об ошибках. В аргументе *limit* передается количество возвращаемых записей.

`extract_stack([frame [, limit]])`

Извлекает ту же трассировочную информацию, что и функция `print_stack()`, но извлекает ее из кадра стека *frame*. Если аргумент *frame* опущен, используется кадр стека вызывающей функции. В аргументе *limit* передается количество возвращаемых записей.

`format_list(list)`

Форматирует трассировочную информацию перед выводом. В аргументе *list* передается список кортежей, возвращаемый функцией `extract_tb()` или `extract_stack()`.

`format_exception_only(type, value)`

Форматирует информацию об исключении перед выводом.

`format_exception(type, value, traceback [, limit])`

Форматирует трассировочную информацию и информацию об исключении перед выводом.

`format_tb(traceback [, limit])`

То же, что и `format_list(extract_tb(traceback, limit))`.

`format_stack([frame [, limit]])`

То же, что и `format_list(extract_stack(frame, limit))`.

`tb_lineno(traceback)`

Возвращает номер строки, установленной в объекте с трассировочной информацией.

Модуль types

В модуле `types` объявляются имена встроенных типов, которые соответствуют функциям, модулям, генераторам, кадрам стека и другим элемен-

там программ. Имена, объявленные в этом модуле, часто используются при обращении к встроенной функции `isinstance()` и в других операциях над типами.

Переменная	Описание
`BuiltinFunctionType`	Тип встроенных функций
`CodeType`	Тип объектов с программным кодом
`FrameType`	Тип кадра стека выполняемых объектов
`FunctionType`	Тип пользовательских функций и `lambda`-выражений
`GeneratorType`	Тип объектов генераторов и итераторов
`GetSetDescriptorType`	Тип объектов дескрипторов методов `getset`
`LambdaType`	Альтернативное имя типа `FunctionType`
`MemberDescriptorType`	Тип объектов дескрипторов атрибутов
`MethodType`	Тип методов пользовательских классов
`ModuleType`	Тип модулей
`TracebackType`	Тип объектов с трассировочной информацией

Большинство типов объектов, перечисленных выше, могут играть роль конструкторов для создания объектов этих типов. В описании ниже представлены параметры, которые передаются при создании функций, модулей, объектов с программным кодом и методов. Подробнее об атрибутах созданных объектов и об аргументах, которые должны передаваться функциям, описываемым ниже, рассказывается в главе 3.

`FunctionType(`*`code`*`, `*`globals`*` [, `*`name`*` [, `*`defargs`*` [, `*`closure`*`]]])`

Создает новый объект функции.

`CodeType(`*`argcount`*`, `*`nlocals`*`, `*`stacksize`*`, `*`flags`*`, `*`codestring`*`, `*`constants`*`, `*`names`*`, `*`varnames`*`, `*`filename`*`, `*`name`*`, `*`firstlineno`*`, `*`lnotab`*` [, `*`freevars`*` [, `*`cellvars`*` ]])`

Создает новый объект с программным кодом.

`MethodType(`*`function`*`, `*`instance`*`, `*`class`*`)`

Создает новый связанный метод объекта.

`ModuleType(`*`name`*` [, `*`doc`*`])`

Создает новый объект модуля.

Примечания

- Модуль `types` не должен использоваться для ссылки на встроенные типы объектов, такие как целые числа, списки или словари. В Python 2 модуль `types` объявляет и другие имена типов, такие как `IntType` и `DictType`. Однако эти имена являются всего лишь псевдонимами имен встроенных типов `int` и `dict`. В новых программах следует использовать имена встроенных типов, потому что в Python 3 модуль `types` теперь содержит только перечисленные выше имена типов.

Модуль warnings

Модуль warnings содержит функции, используемые для фильтрации и вывода предупреждений. В отличие от исключений, предупреждения служат для того, чтобы оповещать пользователя о потенциальных проблемах, но без возбуждения исключений и прерывания выполнения программы. Чаще всего модуль warnings применяется, чтобы информировать пользователей об использовании нерекомендуемых особенностей языка, которые могут не поддерживаться будущими версиями Python. Например:

```
>>> import regex
__main__:1: DeprecationWarning: the regex module is deprecated; use the re
(Перевод: использовать модуль regex не рекомендуется, пользуйтесь модулем re)
module
>>>
```

Подобно исключениям, предупреждения организованы в иерархию классов, которая описывает основные категории предупреждений. Ниже перечислены категории, поддерживаемые в настоящий момент:

Объект предупреждения	Описание
Warning	Базовый класс всех типов предупреждений
UserWarning	Предупреждение, определяемое пользователем
DeprecationWarning	Предупреждение об использовании нерекомендуемой особенности
SyntaxWarning	Потенциальная проблема с синтаксисом
RuntimeWarning	Потенциальная проблема времени выполнения
FutureWarning	Предупреждение о том, что семантика определенной особенности изменится в будущей версии

Все эти классы доступны в модуле __builtin__, так же как и модуль exceptions. Кроме того, все они являются экземплярами класса Exception. Это позволяет легко преобразовывать предупреждения в исключения.

Предупреждения возбуждаются с помощью функции warn(). Например:

```
warnings.warn("не рекомендуется использовать особенность X.")
warnings.warn("работоспособность особенности Y может быть нарушена.",
              RuntimeWarning)
```

При необходимости предупреждения можно фильтровать. Фильтрация может использоваться для изменения порядка вывода предупреждений, подавления вывода предупреждений или преобразования предупреждений в исключения. Для добавления фильтра для определенного типа предупреждения используется функция filterwarnings(). Например:

```
warnings.filterwarnings(action="ignore",
                        message=".*regex.*",
                        category=DeprecationWarning)
import regex     # Предупреждение не появится
```

Ограниченную фильтрацию можно также выполнять с помощью параметра -W командной строки интерпретатора. Например:

```
% python -Wignore:the\ regex:DeprecationWarning
```

В модуле warnings объявлены следующие функции:

warn(*message*[, *category*[, *stacklevel*]])

Возбуждает предупреждение. В аргументе *message* передается строка с текстом предупреждения, в аргументе *category* передается класс предупреждения (такой как DeprecationWarning), а в аргументе *stacklevel* передается целое число, определяющее кадр стека, в котором должно возбуждаться предупреждение. По умолчанию в аргументе *category* передается класс UserWarning, а в аргументе *stacklevel* – число 1.

warn_explicit(*message*, *category*, *filename*, *lineno*[, *module*[, *registry*]])

Низкоуровневая версия функции warn(). Аргументы *message* и *category* имеют тот же смысл, что и в функции warn(). В аргументах *filename*, *lineno* и *module* явно определяется местоположение предупреждения. В аргументе *registry* передается объект, представляющий все активные фильтры. Если аргумент *registry* опущен, вывод предупреждения не подавляется.

showwarning(*message*, *category*, *filename*, *lineno*[, *file*])

Выводит предупреждение *message* в файл *file*. Если аргумент *file* опущен, предупреждение выводится в поток sys.stderr.

formatwarning(*message*, *category*, *filename*, *lineno*)

Создает форматированную строку, которая выводится при возбуждении предупреждения.

filterwarnings(*action*[, *message*[, *category*[, *module*[, *lineno*[, *append*]]]]])

Добавляет новый элемент в список фильтров. В аргументе *action* передается строка 'error', 'ignore', 'always', 'default', 'once' или 'module'. Ниже приводится описание каждого из этих значений:

Действие	Описание
'error'	Преобразует предупреждение в исключение
'ignore'	Подавляет вывод предупреждения
'always'	Всегда выводить текст предупреждения
'default'	Предупреждение выводится однократно для каждого из местоположений, где оно встречается
'module'	Предупреждение выводится однократно для каждого из модулей, где оно встречается
'once'	Предупреждение выводится однократно, независимо от того, где оно возникло

В аргументе *message* передается строка с регулярным выражением, которое будет сопоставляться с текстом сообщения. В аргументе *category* передается класс предупреждения, такой как DeprecationError. В аргументе *module*

передается строка с регулярным выражением, которое будет сопоставляться с именем модуля. В аргументе *lineno* передается номер строки или 0, где нулевое значение соответствует всем строкам. Аргумент *append* определяет, должен ли фильтр добавляться в конец списка всех фильтров (то есть он будет проверяться последним). По умолчанию новые фильтры добавляются в начало списка фильтров. Если какой-либо из аргументов опущен, ему по умолчанию присваивается значение, соответствующее всем предупреждениям.

`resetwarnings()`

Сбрасывает все фильтры предупреждений. При этом уничтожаются все фильтры, созданные ранее вызовом функции `filterwarnings()`, а также заданные в параметре `-W`.

Примечания

- Список активных фильтров находится в переменной `warnings.filters`.
- Когда предупреждение преобразуется в исключение, категория предупреждения становится типом исключения. Например, при преобразовании предупреждения `DeprecationWarning` в ошибку будет возбуждено исключение `DeprecationWarning`.
- Определить фильтры в командной строке можно с помощью параметра `-W`. В общем виде этот параметр имеет следующий формат:

  ```
  -Waction:message:category:module:lineno
  ```

  где каждый аргумент имеет тот же смысл, что и в функции `filterwarning()`. Однако в данном случае поля *message* и *module* определяются не как регулярные выражения, а как подстроки из начала текста предупреждения и имени модуля, предназначенные для фильтрации.

Модуль weakref

Модуль `weakref` обеспечивает поддержку слабых ссылок. В обычном случае сохранение ссылки на объект приводит к увеличению счетчика ссылок, что препятствует уничтожению объекта, пока значение счетчика не достигнет нуля. Слабая ссылка позволяет обращаться к объекту, не увеличивая его счетчик ссылок. Это может пригодиться в некоторых случаях, когда требуется организовать управление объектами необычными способами. Например, в объектно-ориентированных программах, если требуется реализовать отношения между объектами, как в шаблоне проектирования «Наблюдатель» (Observer), чтобы избежать циклических ссылок, можно использовать слабые ссылки. Пример такой реализации приводится в разделе «Управление памятью объектов» главы 7.

Создается слабая ссылка с помощью функции `weakref.ref()`, как показано ниже:

```
>>> class A: pass
>>> a = A()
>>> ar = weakref.ref(a) # Создаст слабую ссылку на a
```

```
>>> print ar
<weakref at 0x135a24; to 'instance' at 0x12ce0c>
```

После получения слабой ссылки доступ к оригинальному объекту можно будет получить простым обращением к слабой ссылке, как к функции без аргументов. Если объект все еще существует, функция вернет его. В противном случае она вернет `None`, как признак того, что оригинальный объект больше не существует. Например:

```
>>> print ar()         # Выведет оригинальный объект
<__main__.A instance at 12ce0c>
>>> del a              # Удалит оригинальный объект
>>> print ar()         # Объект a был удален, поэтому теперь возвращается None
None
>>>
```

В модуле `weakref` объявлены следующие функции:

`ref(`*`object`*`[, `*`callback`*`])`

Создает слабую ссылку на объект *object*. В необязательном аргументе *callback* передается функция, которая должна вызываться перед удалением объекта *object*. Эта функция должна принимать единственный аргумент, в котором ей будет передаваться слабая ссылка на объект. На один и тот же объект может указывать сразу несколько слабых ссылок. В этом случае функции *callback* будут вызываться в порядке, обратном порядку создания слабых ссылок – от более свежих к более старым. Доступ к объекту *object* можно получить, обратившись к слабой ссылке как к функции без аргументов. Если оригинальный объект был удален, функция вернет `None`. Функция `ref()` в действительности принадлежит к типу `ReferenceType`, который может использоваться в операциях проверки типов и создания производных классов.

`proxy(`*`object`*`[, `*`callback`*`])`

Создает прокси-объект со слабой ссылкой на заданный объект *object*. Возвращаемый объект в действительности является оберткой вокруг оригинального объекта, обеспечивающей доступ к его атрибутам и методам. Пока будет существовать оригинальный объект, прокси-объект будет имитировать его поведение. Однако после уничтожения оригинального объекта любая попытка обращения к прокси-объекту будет возбуждать исключение `weakref.ReferenceError`, чтобы показать, что оригинальный объект был удален. В аргументе *callback* передается функция обратного вызова, имеющая то же назначение, что и в функции `ref()`. Прокси-объект может иметь тип либо `ProxyType`, либо `CallableProxyType`, в зависимости от того, является ли оригинальный объект вызываемым объектом.

`getweakrefcount(`*`object`*`)`

Возвращает количество слабых ссылок и объектов-оберток, указывающих на объект *object*.

`getweakrefs(`*`object`*`)`

Возвращает список всех слабых ссылок и объектов-оберток, указывающих на объект *object*.

`WeakKeyDictionary([dict])`

Создает словарь, в котором ключи представлены слабыми ссылками. Когда количество обычных ссылок на объект ключа становится равным нулю, соответствующий элемент словаря автоматически удаляется. В необязательном аргументе *dict* передается словарь, элементы которого добавляются в возвращаемый объект типа `WeakKeyDictionary`. Поскольку слабые ссылки могут создаваться только для объектов определенных типов, существует большое число ограничений на допустимые типы объектов ключей. В частности, встроенные строки не могут использоваться в качестве ключей со слабыми ссылками. Однако экземпляры пользовательских классов, объявляющих метод `__hash__()`, могут играть роль ключей. Экземпляры класса `WeakKeyDictionary` имеют два дополнительных метода, `iterkeyrefs()` и `keyrefs()`, которые возвращают слабые ссылки на ключи.

`WeakValueDictionary([dict])`

Создает словарь, в котором значения представлены слабыми ссылками. Когда количество обычных ссылок на объект значения становится равным нулю, соответствующий элемент словаря автоматически удаляется. В необязательном аргументе *dict* передается словарь, элементы которого добавляются в возвращаемый объект типа `WeakValueDictionary`. Экземпляры класса `WeakValueDictionary` имеют два дополнительных метода, `itervaluerefs()` и `valuerefs()`, которые возвращают слабые ссылки на значения.

`ProxyTypes`

Это кортеж (`ProxyType, CallableProxyType`), который может использоваться для проверки на принадлежность объекта к одному из типов прокси-объектов, создаваемых функцией `proxy()`, например `isinstance(object, ProxyTypes)`.

Пример

Одно из применений слабых ссылок – создание кэшей недавно вычисленных результатов. Например, когда функции требуется достаточно продолжительное время для вычисления результата, имеет смысл кэшировать эти результаты и повторно использовать их, пока они используются где-нибудь в приложении. Например:

```
_resultcache = { }
def foocache(x):
    if resultcache.has_key(x):
        r = _resultcache[x]()   # Получение слабой ссылки и ее разыменование
        if r is not None: return r
    r = foo(x)
    _resultcache[x] = weakref.ref(r)
    return r
```

Примечания

- Возможность создания слабых ссылок поддерживается только экземплярами классов, функциями, методами, множествами, фиксированными множествами, объектами файлов, генераторами, объектами ти-

пов и некоторыми объектами типов, объявленными в библиотечных модулях (например, сокетами, массивами и шаблонами регулярных выражений). Встроенные функции и большинство встроенных типов, таких как списки, словари, строки и числа, не позволяют создавать слабые ссылки на них.

- Если в программе потребуется выполнять итерации по элементам объектов `WeakKeyDictionary` или `WeakValueDictionary`, следует сделать все возможное, чтобы обеспечить неизменность размеров словаря, так как это может приводить к странным побочным эффектам, например к мистическому исчезновению элементов словаря без всяких видимых причин.
- Если в процессе выполнения функции обратного вызова, зарегистрированной с помощью функции `ref()` или `proxy()`, будет возбуждено исключение, оно будет выведено в поток стандартного вывода сообщений об ошибках и проигнорировано.
- Слабые ссылки являются хешируемыми, если оригинальный объект также является хешируемым. Кроме того, слабая ссылка будет продолжать хранить свое значение хеша даже после удаления оригинального объекта, при условии, что оно будет вычислено, пока объект существует.
- Слабые ссылки могут проверяться на равенство, но для них не существует понятий «больше» или «меньше». Если оригинальные объекты продолжают существовать, ссылки будут определяться как равные, если оригинальные объекты имеют одно и то же значение или ссылки указывают на один и тот же объект.

14

Математика

В этой главе описываются модули, предназначенные для выполнения различных видов математических операций. Дополнительно здесь описывается модуль `decimal`, обеспечивающий обобщенную поддержку дробных десятичных чисел с плавающей точкой.

Модуль decimal

Тип данных `float` в языке Python представлен в виде чисел с плавающей точкой двойной точности (как определено стандартом IEEE 754). Одна из особенностей этого формата состоит в том, что при его использовании невозможно точно представить десятичные дробные значения, такие как 0.1, так как в этом формате самым близким значением является 0.10000000000000001. Эта погрешность свойственна всем вычислениям, где участвуют числа с плавающей точкой, и иногда она может приводить к неожиданным результатам (например, выражение `3 * 0.1 == 0.3` возвращает значение `False`).

Модуль `decimal` предоставляет реализацию стандарта IBM General Decimal Arithmetic Standard, который обеспечивает возможность точного представления дробных десятичных значений. Кроме того, он обеспечивает полный контроль над математической точностью, количеством значимых цифр и способом округления. Эти возможности могут пригодиться при организации взаимодействий с внешними системами, в которых свойства дробных десятичных чисел определяются точно. Например, с помощью этого модуля можно создавать программы на языке Python, которые должны взаимодействовать с бизнес-приложениями.

В модуле `decimal` объявляется два основных типа данных: тип `Decimal`, представляющий дробные десятичные числа, и тип `Context`, представляющий различные параметры, касающиеся вычислений, такие как точность и обработка ошибок округления. Ниже приводится несколько простых примеров, иллюстрирующих основные приемы работы с модулем:

```
import decimal
x = decimal.Decimal('3.4')    # Создать несколько дробных десятичных чисел
y = decimal.Decimal('4.5')

# Выполнить некоторые вычисления, используя контекст по умолчанию
a = x * y            # a = decimal.Decimal('15.30')
b = x / y            # b = decimal.Decimal('0.755555555555555555555555556')

# Изменить точность и повторить вычисления
decimal.getcontext().prec = 3
c = x * y            # c = decimal.Decimal('15.3')
d = x / y            # d = decimal.Decimal('0.756')

# Изменить точность только для одного блока инструкций
with decimal.localcontext(decimal.Context(prec=10)):
    e = x * y        # e = decimal.Decimal('15.30')
    f = x / y        # f = decimal.Decimal('0.7555555556')
```

Объекты Decimal

Числа типа `Decimal` представлены следующим классом:

`Decimal([`*value* `[, ` *context*`]])`

В аргументе *value* передается значение числа, которое может быть целым числом, строкой, содержащей значение дробного десятичного числа, такое как '4.5', или кортежем (*sign*, *digits*, *exponent*). Если значение представлено кортежем, значение 0 в поле *sign* означает положительное число, 1 – отрицательное; в поле *digits* передается кортеж цифр в виде целых чисел и в поле *exponent* передается целочисленная экспонента. Для обозначения положительной или отрицательной бесконечности могут использоваться специальные строковые значения `'Infinity'`, `'-Infinity'`, а для обозначения «не числа» (Not a Number, NaN) – строки `'NaN'` и `'sNaN'`. Значение `'sNaN'` – это разновидность значения «не числа», которая вызывает исключение при попытке использовать его в вычислениях. Обычный объект типа `float` не может использоваться в качестве начального значения, потому что это значение может быть неточным (что идет вразрез с назначением обращения к типу `decimal`). В аргументе *context* передается объект класса `Context`, который будет описан ниже. Аргумент *context* определяет, что должно произойти, если начальное значение не является допустимым числом, – будет возбуждено исключение или возвращено значение NaN.

Следующие примеры демонстрируют, как создавать различные дробные десятичные числа:

```
a = decimal.Decimal(42)               # Создаст Decimal("42")
b = decimal.Decimal("37.45")          # Создаст Decimal("37.45")
c = decimal.Decimal((1,(2,3,4,5),-2)) # Создаст Decimal("-23.45")
d = decimal.Decimal("Infinity")
e = decimal.Decimal("NaN")
```

Объекты `Decimal` относятся к разряду неизменяемых и обладают всеми обычными свойствами чисел встроенных типов `int` и `float`. Они могут также использоваться в качестве ключей словаря, помещаться в множества, сортироваться и так далее. В большинстве случаев манипулирование объ-

ектами типа Decimal производится с помощью стандартных математических операторов языка Python. Помимо этого ниже представлены методы, которые могут использоваться для выполнения некоторых стандартных математических операций. Все методы принимают необязательный аргумент *context*, управляющий точностью, округлением и другими аспектами вычислений. Если этот аргумент опущен, используется текущий контекст.

Метод	Описание
x.exp([*context*])	Натуральная степень *e* ** *d*
x.fma(*y*, *z* [, *context*])	*x* * *y* + *z* без округления результата *x***y*
x.ln([*context*])	Натуральный логарифм (по основанию e) числа *x*
x.log10([*context*])	Десятичный логарифм числа *x*
x.sqrt([*context*])	Корень квадратный из числа *x*

Объекты Context

Управление различными аспектами дробных десятичных чисел, такими как необходимость округления и точность, производится с помощью объекта типа Context:

Context(*prec*=None, *rounding*=None, *traps*=None, *flags*=None, *Emin*=None, *Emax*=None, *capitals*=1)

Создает новый контекст для работы с дробными десятичными числами. Параметры контекста должны определяться в виде именованных аргументов с указанными именами. В аргументе *prec* передается целое число, определяющее количество цифр после десятичной точки, аргумент *rounding* определяет порядок округления, а в аргументе *traps* передается список сигналов, которые возбуждают исключения в различных обстоятельствах (например, при попытке выполнить деление на ноль). В аргументе *flags* передается список сигналов, свидетельствующих о начальном состоянии контекста (например, переполнение). Обычно аргумент *flags* не указывается. В аргументах *Emin* и *Emax* передаются целые числа, определяющие минимальное и максимальное значения экспоненты соответственно. В аргументе *capitals* передается логический флаг, указывающий, какой символ, 'E' или 'e', должен использоваться для обозначения экспоненты. По умолчанию имеет значение 1 ('E').

Обычно новые объекты класса Context не создаются непосредственно. Для этого используются функция getcontext() или localcontext(), возвращающая текущий объект Context. Затем этот объект изменяется в соответствии с требованиями. Примеры использования этих функций приводятся ниже в этом же разделе. Однако, чтобы лучше понять эти примеры, необходимо подробнее описать эти параметры контекста.

Порядок округления определяется аргументом *rounding*, который может иметь одно из следующих значений:

Константа	Описание
ROUND_CEILING	Округление в сторону положительной бесконечности. Например, число 2.52 будет округлено до 2.6, а число –2.58 – до –2.5.
ROUND_DOWN	Округление в сторону нуля. Например, число 2.58 будет округлено до 2.5, а число –2.58 – до -2.5.
ROUND_FLOOR	Округление в сторону отрицательной бесконечности. Например, число 2.52 будет округлено до 2.5, а число –2.58 – до –2.6.
ROUND_HALF_DOWN	Округление в сторону от нуля, если округляемая часть больше половины последнего значимого разряда, в противном случае округление будет выполнено в сторону нуля. Например, число 2.58 будет округлено до 2.6, число 2.55 будет округлено до 2.5, а число –2.58 – до –2.6.
ROUND_HALF_EVEN	То же, что и ROUND_HALF_DOWN, только если округляемая часть равна точно половине последнего значимого разряда, результат округляется вниз, если предыдущая цифра четная, и вверх – если предыдущая цифра нечетная. Например, число 2.65 будет округлено до 2.6, число 2.55 также будет округлено до 2.6.
ROUND_HALF_UP	То же, что и ROUND_HALF_DOWN, только если округляемая часть равна точно половине последнего значимого разряда, результат округляется в сторону от нуля. Например, число 2.55 будет округлено до 2.6, а число –2.55 до –2.6.
ROUND_UP	Округление в сторону от нуля. Например, число 2.52 будет округлено до 2.6, а число –2.52 – до –2.6.
ROUND_05UP	Округление в сторону от нуля, если последний значимый разряд содержит цифру 0 или 5, в противном случае округление выполняется в сторону нуля. Например, число 2.54 будет округлено до 2.6, число 2.64 также будет округлено до 2.6.

В аргументах *traps* и *flags* конструктору Context() передаются списки сигналов. Сигнал представляет тип арифметического исключения, которое может возникнуть в процессе вычислений. Если аргументы *listed* и *traps* опущены, все сигналы просто игнорируются. В противном случае возбуждается исключение. Возможные сигналы перечислены ниже:

Сигнал	Описание
Clamped	Экспонента была откорректирована в соответствии с допустимым диапазоном.
DivisionByZero	Деление небесконечного числа на 0.
Inexact	Погрешность округления.
InvalidOperation	Выполнена недопустимая операция.

Сигнал	Описание
Overflow	После округления экспонента превысила значение *Emax*. Также генерирует сигналы Inexact и Rounded.
Rounded	Округление выполнено. Может появиться, если при округлении точность представления числа не пострадала (например, при округлении «1.00» до «1.0»).
Subnormal	Перед округлением экспонента была меньше значения *Emin*.
Underflow	Потеря значащих разрядов числа. Результат операции был округлен до 0. Также генерирует сигналы Inexact и Subnormal.

Этим сигналам соответствуют исключения, которые могут использоваться для проверки на наличие ошибок. Например:

```
try:
    x = a/b
except decimal.DivisionByZero:
    print "Деление на ноль"
```

Как и исключения, сигналы организованы в иерархию:

```
ArithmeticError (встроенное исключение)
    DecimalException
        Clamped
        DivisionByZero
    Inexact
        Overflow
        Underflow
    InvalidOperation
    Rounded
        Overflow
        Underflow
    Subnormal
        Underflow
```

Сигналы Overflow и Underflow присутствуют в этой иерархии в нескольких местах, потому что эти сигналы могут вызывать появление родительского сигнала (например, сигнал Underflow влечет за собой появление сигнала Subnormal). Кроме того, сигнал decimal.DivisionByZero является производным от встроенного исключения DivisionByZero.

Во многих случаях арифметические сигналы попросту игнорируются. Например, в процессе вычислений может возникать погрешность округления, но не приводить к возбуждению исключения. Имена сигналов можно использовать для проверки флагов, указывающих на состояние вычислений. Например:

```
ctxt = decimal.getcontext() # Получить текущий контекст
x = a + b
if ctxt.flags[Rounded]:
    print "Результат был округлен!"
```

После установки флаги сохраняют свое состояние, пока не будет вызван метод clear_flags(). Благодаря этому можно выполнить целую серию вычислений и проверить ошибки только в конце.

Значения параметров существующего объекта *c* типа Context можно получить с помощью следующих атрибутов и методов:

c.capitals

Флаг может иметь значение 1 или 0 и определяет, какой символ, E или e, будет использоваться для обозначения экспоненты.

c.Emax

Целое число, определяющее максимальное значение экспоненты.

c.Emin

Целое число, определяющее минимальное значение экспоненты.

c.prec

Целое число, определяющее количество знаков после десятичной точки.

c.flags

Словарь, содержащий текущие значения флагов, соответствующих сигналам. Например, обращение к элементу *c*.flags[Rounded] вернет текущее значение флага, соответствующего сигналу Rounded.

c.rounding

Действующее правило округления. Например ROUND_HALF_EVEN.

c.traps

Словарь, содержащий значения True/False, соответствующие сигналам, которые должны вызывать исключения. Например, элемент *c*.traps[DivisionByZero] обычно имеет значение True, тогда как *c*.traps[Rounded] — False.

c.clear_flags()

Сбрасывает все флаги (очищает словарь *c*.flags).

c.copy()

Возвращает копию контекста *c*.

c.create_decimal(*value*)

Создает новый объект Decimal, используя контекст *c*. Это может пригодиться, когда потребуется создавать числа, точность представления и правила округления для которых должны отличаться от установленных по умолчанию.

Функции и константы

В модуле decimal объявлены следующие функции и константы.

getcontext()

Возвращает текущий контекст дробных десятичных чисел. Каждый поток выполнения имеет свой собственный контекст, поэтому данная функция возвращает контекст для вызывающего потока выполнения.

`localcontext([c])`

Создает менеджера контекста, который устанавливает контекст *c* дробных десятичных чисел в качестве текущего для инструкций, находящихся в теле инструкции `with`. При вызове без аргумента создает копию текущего контекста. Ниже приводится пример использования этой функции, с помощью которой временно устанавливается точность до пяти знаков после десятичной точки для последовательности инструкций:

```
with localcontext() as c:
    c.prec = 5
    инструкции
```

`setcontext(c)`

Устанавливает контекст *c* дробных десятичных чисел в качестве текущего для вызывающего потока выполнения.

`BasicContext`

Предопределенный контекст с точностью до девяти знаков после десятичной точки. Использует правило округления `ROUND_HALF_UP`; параметр `Emin` имеет значение -999999999; параметр `Emax` имеет значение 999999999; разрешены все сигналы, кроме `Inexact`, `Rounded` и `Subnormal`.

`DefaultContext`

Контекст по умолчанию, который используется при создании нового контекста (то есть значения параметров этого контекста используются как значения по умолчанию для параметров нового контекста). Определяет точность до 28 знаков после десятичной точки; округление `ROUND_HALF_EVEN`; включает флаги `Overflow`, `InvalidOperation` и `DivisionByZero`.

`ExtendedContext`

Предопределенный контекст с точностью до девяти знаков после десятичной точки. Использует правило округления `ROUND_HALF_EVEN`; параметр `Emin` имеет значение -999999999; параметр `Emax` имеет значение 999999999; все сигналы запрещены. Никогда не возбуждает исключения, но в результате операций может возвращаться значение `NaN` или `Infinity`.

`Inf`

То же, что и `Decimal("Infinity")`.

`negInf`

То же, что и `Decimal("-Infinity")`.

`NaN`

То же, что и `Decimal("NaN")`.

Примеры

Ниже приводится несколько примеров, иллюстрирующих основы работы с дробными десятичными числами:

```
>>> a = Decimal("42.5")
>>> b = Decimal("37.1")
```

```
>>> a + b
Decimal("79.6")
>>> a / b
Decimal("1.14555256064690026954177897 6")
>>> divmod(a,b)
(Decimal("1"), Decimal("5.4"))
>>> max(a,b)
Decimal("42.5")
>>> c = [Decimal("4.5"), Decimal("3"), Decimal("1.23e3")]
>>> sum(c)
Decimal("1237.5")
>>> [10*x for x in c]
[Decimal("45.0"), Decimal("30"), Decimal("1.230e4")]
>>> float(a)
42.5
>>> str(a)
'42.5'
```

Далее приводится пример изменения параметров контекста:

```
>>> getcontext().prec = 4
>>> a = Decimal("3.4562384105")
>>> a
Decimal("3.4562384105")
>>> b = Decimal("5.6273833")
>>> getcontext().flags[Rounded]
0
>>> a + b
9.084
>>> getcontext().flags[Rounded]
1
>>> a / Decimal("0")
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
decimal.DivisionByZero: x / 0
>>> getcontext().traps[DivisionByZero] = False
>>> a / Decimal("0")
Decimal("Infinity")
```

Примечания

- Объекты `Decimal` и `Context` обладают большим количеством методов, связанных с низкоуровневыми особенностями, которые имеют отношение к представлению и выполнению операций над дробными десятичными числами. Они не были описаны здесь, потому что знакомство с ними не является необходимым условием использования данного модуля в простых случаях. Однако вы сможете найти подробное их описание в электронной документации, по адресу: *http://docs.python.org/library/decimal.html*.
- Контекст дробных десятичных чисел является уникальным для каждого потока выполнения. Изменения в контексте будут оказывать влияние только в данном потоке, но не в других.

- Чтобы сигнализировать о появлении значения, не являющегося числом, можно использовать специальное число `Decimal("sNaN")`. Это число никогда не генерируется встроенными функциями. Однако если оно появится в вычислениях, будет выведено сообщение об ошибке. Его можно использовать, чтобы известить пользователя об ошибках, когда ошибочные вычисления должны быть прерваны и ошибки не могут быть проигнорированы. Например, функция, где возникла ошибка, может возвращать `sNaN` в качестве результата.
- Значение 0 может быть положительным или отрицательным (то есть `Decimal(0)` и `Decimal("-0")`). Однако при сравнивании эти значения попрежнему считаются равными.
- Этот модуль, вероятно, мало подходит для программирования высокопроизводительных научных вычислений из-за большого объема операций, выполняемых в процессе вычислений. Кроме того, в подобных приложениях дробные десятичные числа имеют слишком мало преимуществ перед числами с плавающей точкой в двоичном формате.
- Подробное обсуждение форматов представления чисел с плавающей точкой и анализ ошибок выходит далеко за рамки данной книги. За дополнительными подробностями заинтересованным читателям следует обратиться к книге по численному анализу. Кроме того, нелишним будет ознакомиться со статьей «What Every Computer Scientist Should Know About Floating-Point Arithmetic» («Что должен знать каждый специалист в области вычислительных машин об арифметике с плавающей точкой») Дэвида Голдберга (David Goldberg), опубликованной в журнале «Computing Surveys», издательство Association for Computing Machinery, март 1991 года (эту статью легко отыскать в Интернете, поискав по заголовку).[1]
- Спецификация «IBM General Decimal Arithmetic Specification» содержит массу дополнительной информации и также легко может быть найдена в Интернете с помощью поисковых систем.

Модуль fractions

Модуль `fractions` содержит определение класса `Fraction`, который является представлением рациональных чисел. Создать экземпляр класса можно с помощью конструктора, тремя разными способами:

`Fraction([`*`numerator`* `[,`*`denominator`*`]])`

Создает новое рациональное число. Числитель (*`numerator`*) и знаменатель (*`denominator`*) являются целыми числами и по умолчанию принимаются равными `0` и `1` соответственно.

[1] Перевода этой статьи на русский язык в Интернете нет, но можно порекомендовать отличные статьи: *http://www.vbstreets.ru/VB/Articles/66541.aspx* и *http://www.ibm.com/developerworks/ru/library/j-jtp0114/*. – *Прим. перев.*

`Fraction(`*fraction*`)`

Если в аргументе *fraction* передается экземпляр `numbers.Rational`, создается новое рациональное число с тем же значением, что и *fraction*.

`Fraction(`*s*`)`

Если в аргументе *s* передается строка, содержащая дробь, такую как "3/7" или "-4/7", создается дробь с теми же значениями числителя и знаменателя. Если в аргументе *s* передается строка с изображением дробного десятичного числа, например "1.25", создается дробь, соответствующая этому значению (то есть `Fraction(5,4)`).

Ниже перечислены методы класса, которые могут использоваться для создания экземпляров класса `Fraction` из объектов других типов:

`Fraction.from_float(`*f*`)`

Создает дробь, представляющую точное значение числа *f* с плавающей точкой.

`Fraction.from_decimal(`*d*`)`

Создает дробь, представляющую точное значение числа *d* типа `Decimal`.

Ниже приводится несколько примеров использования этих функций:

```
>>> f = fractions.Fraction(3,4)
>>> f
Fraction(3, 4)
>>> g = fractions.Fraction("1.75")
>>> g
Fraction(7, 4)
>>> h = fractions.Fraction.from_float(3.1415926)
Fraction(3537118815677477, 1125899906842624)
>>>
```

Экземпляр *f* класса `Fraction` поддерживает все обычные математические операции. Значения числителя и знаменателя хранятся в атрибутах *f*`.numerator` и *f*`.denominator` соответственно. Кроме того, определены следующие методы:

f`.limit_denominator([`*max_denominator*`])`

Возвращает дробь, ближайшую к *f*. В аргументе *max_denominator* передается наибольший возможный числитель. По умолчанию используется значение 1000000.

Ниже приводится несколько примеров использования экземпляров класса `Fraction` (задействованы значения, созданные в предыдущем примере):

```
>>> f + g
Fraction(5, 2)
>>> f * g
Fraction(21, 16)
>>> h.limit_denominator(10)
Fraction(22, 7)
>>>
```

Кроме того, в модуле `fractions` определена единственная функция:

`gcd(a, b)`

Вычисляет наибольший общий делитель целых чисел *a* и *b*. Результат имеет тот же знак, что и число *b*, если оно не равно нулю; и знак числа *a* – в противном случае.

Модуль math

В модуле `math` определены следующие стандартные математические функции. Эти функции оперируют целыми числами и числами с плавающей точкой, но они не будут работать с комплексными числами (для выполнений подобных операций над комплексными числами можно использовать отдельный модуль `cmath`). Все функции возвращают число с плавающей точкой. Все тригонометрические функции оперируют угловыми величинами, выраженными в радианах.

Функция	Описание
`acos(x)`	Возвращает арккосинус числа *x*.
`acosh(x)`	Возвращает гиперболический арккосинус числа *x*.
`asin(x)`	Возвращает арксинус числа *x*.
`asinh(x)`	Возвращает гиперболический арксинус числа *x*.
`atan(x)`	Возвращает арктангенс числа *x*.
`atan2(y,x)`	Возвращает арктангенс выражения (*y* / *x*).
`atanh(x)`	Возвращает гиперболический арктангенс числа *x*.
`ceil(x)`	Возвращает округленное до наибольшего целого значение числа *x*.
`copysign(x,y)`	Возвращает *x* с тем же знаком, что и *y*.
`cos(x)`	Возвращает косинус числа *x*.
`cosh(x)`	Возвращает гиперболический косинус числа *x*.
`degrees(x)`	Преобразует число *x* радианов в градусы.
`radians(x)`	Преобразует число *x* *градусов в радианы*.
`exp(x)`	Возвращает `e ** x`.
`fabs(x)`	Возвращает абсолютное значение *x*.
`factorial(x)`	Возвращает факториал *x*.
`floor(x)`	Возвращает округленное до наименьшего целого значение числа *x*.
`fmod(x,y)`	Возвращает `x % y`, как вычисляет функция `fmod()` в языке C.
`frexp(x)`	Возвращает положительное значение мантиссы и экспоненту числа *x* в виде кортежа.

(продолжение)

Функция	Описание
`fsum(s)`	Возвращает сумму значений с плавающей точкой в итерируемой последовательности *s*. Дополнительные подробности приводятся в разделе «Примечания» ниже.
`hypot(x,y)`	Возвращает эвклидово расстояние, `sqrt(x * x + y * y)`.
`isinf(x)`	Возвращает `True`, если *x* имеет значение бесконечности.
`isnan(x)`	Возвращает `True`, если *x* имеет значение `NaN`.
`ldexp(x,i)`	Возвращает `x * (2 ** i)`.
`log(x [, base])`	Возвращает логарифм числа *x* по основанию *base*. Если аргумент *base* опущен, вычисляется натуральный логарифм.
`log10(x)`	Возвращает десятичный логарифм числа *x*.
`log1p(x)`	Возвращает натуральный логарифм выражения *x*+1.
`modf(x)`	Возвращает дробную и целую части числа *x* в виде кортежа. Оба значения имеют тот же знак, что и число *x*.
`pow(x,y)`	Возвращает `x ** y`.
`sin(x)`	Возвращает синус числа *x*.
`sinh(x)`	Возвращает гиперболический синус числа *x*.
`sqrt(x)`	Возвращает квадратный корень числа *x*.
`tan(x)`	Возвращает тангенс числа *x*.
`tanh(x)`	Возвращает гиперболический тангенс числа *x*.
`trunc(x)`	Усекает дробную часть числа *x*.

В модуле `math` определены следующие константы:

Константа	Описание
`pi`	Математическая константа – число π.
`e`	Математическая константа – число *e*.

Примечания

- Значения с плавающей точкой `+inf`, `-inf` и `nan` могут создаваться вызовом функции `float()` со строковым аргументом, например: `float("inf")`, `float("-inf")` или `float("nan")`.
- Функция `math.fsum()` обеспечивает более высокую точность, чем встроенная функция `sum()`, потому что в ней используется другой алгоритм, в котором предпринимается попытка избежать ошибок, обусловленных эффектом взаимного уничтожения. Возьмем в качестве примера последовательность `s = [1, 1e100, -1e100]`. Если попытаться вычислить сумму ее элементов вызовом функции `sum(s)`, в результате будет полу-

чено значение 0.0 (потому что значение 1 будет потеряно при сложении с большим числом 1e100). Однако вызов `math.sum(s)` вернет правильный результат 1.0. Алгоритм, используемый в функции `math.fsum()`, описывается в статье «Adaptive Precision Floating-Point Arithmetic and Fast Robust Geometric Predicates» (Адаптивная точность в арифметике с плавающей точкой и быстрые устойчивые геометрические предикаты) Джонатана Ричарда Шевчука (Jonathan Richard Shewchuk) в техническом отчете колледжа информатики университета Карнеги-Меллона (Carnegie Mellon University. School of Computer Science Technical Report) CMU-CS-96-140, 1996.

Модуль numbers

В модуле `numbers` определено несколько абстрактных базовых классов, которые могут использоваться для создания чисел различных типов. Числовые классы организованы в иерархию, на каждом уровне которой последовательно добавляются дополнительные возможности.

`Number`

Класс, находящийся на вершине иерархии числовых классов.

`Complex`

Класс, представляющий комплексные числа. Числа этого вида состоят из действительной и мнимой частей и имеют атрибуты `real` и `imag`. Является производным от класса `Number`.

`Real`

Класс, представляющий вещественные числа. Является производным от класса `Complex`.

`Rational`

Класс, представляющий рациональные дроби. Числа этого вида состоят из числителя и знаменателя и имеют атрибуты `numerator` и `denominator`. Является производным от класса `Real`.

`Integral`

Класс, представляющий целые числа. Является производным от класса `Rational`.

Классы в этом модуле не предназначены для создания экземпляров. Их основная цель – обеспечить возможность проверки типов различных значений. Например:

```
if isinstance(x, numbers.Number)   # объект x является числом любого типа
    инструкции
if isinstance(x, numbers.Integral) # объект x является целым числом
    инструкции
```

Если какая-либо из этих проверок типа вернет `True`, это означает, что объект *x* совместим со всеми математическими операциями, которые могут выполняться над числами данного типа, и может быть преобразован в один из встроенных типов, такой как `complex()`, `float()` или `int()`.

Кроме того, абстрактные базовые классы могут использоваться в качестве базовых для пользовательских классов чисел. Такое наследование обеспечит не только возможность проверки типов, но и гарантирует реализацию всех обязательных методов. Например:

```
>>> class Foo(numbers.Real): pass
...
>>> f = Foo()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't instantiate abstract class Foo with abstract methods
(Перевод:
Трассировочная информация (самый последний вызов - самый нижний):
  Файл "<stdin>", строка 1, в <модуль>
TypeError: Невозможно создать экземпляр абстрактного класса Foo с абстрактными
методами
)
__abs__, __add__, __div__, __eq__, __float__, __floordiv__, __le__, __lt__,
__mod__, __mul__, __neg__, __pos__, __pow__, __radd__, __rdiv__, __rfloordiv__,
__rmod__, __rmul__, __rpow__, __rtruediv__, __truediv__, __trunc__
>>>
```

Примечания

- Подробнее об абстрактных базовых классах рассказывается в главе 7 «Классы и объектно-ориентированное программирование».
- Более подробную информацию об иерархии типов в этом модуле и их назначении можно найти в предложении по развитию Python (Python Enhancement Proposal, PEP) PEP 3141 (*http://www.python.org/dev/peps/pep-3141*).

Модуль random

Модуль `random` предоставляет различные функции, генерирующие псевдослучайные числа, а также функции, генерирующие псевдослучайные числа в соответствии с различными законами распределения. Большая часть функций в этом модуле опирается на использование функции `random()`, которая с помощью генератора «Вихрь Мерсенна» генерирует псевдослучайные числа, равномерно распределенные по диапазону `[0.0, 1.0)`.

Инициализация

Следующие функции используются для управления состоянием генератора случайных чисел:

`seed([x])`

Инициализирует генератор случайных чисел. При вызове без аргумента или со значением `None` в аргументе *x* генератор инициализируется значением системного времени. В противном случае, если *x* является целым или длинным целым числом, используется значение *x*. Если аргумент *x* не является целым числом, он должен быть хешируемым объектом, а в каче-

стве инициализирующего значения будет использоваться результат вызова функции hash(*x*).

`getstate()`

Возвращает объект, представляющий текущее состояние генератора. Позднее этот объект можно передать функции `setstate()`, чтобы восстановить состояние генератора.

`setstate(`*state*`)`

Восстанавливает состояние генератора случайных чисел из объекта, полученного в результате вызова функции `getstate()`.

`jumpahead(`*n*`)`

Быстро изменяет состояние генератора, как если бы функция `random()` была вызвана *n* раз подряд. Аргумент *n* должен быть целым неотрицательным числом.

Случайные целые числа

Следующие функции используются для получения случайных целых чисел.

`getrandbits(`*k*`)`

Создает длинное целое число, состоящее из *k* случайных битов.

`randint(`*a*`,`*b*`)`

Возвращает случайное целое число *x* в диапазоне *a* <= *x* <= *b*.

`randrange(`*start*`,`*stop* `[,`*step*`])`

Возвращает случайное целое число в диапазоне (*start*,*stop*,*step*). Не включая значение *stop*.

Случайные последовательности

Следующие функции используются для рандомизации последовательностей.

`choice(`*seq*`)`

Возвращает случайный элемент из непустой последовательности *seq*.

`sample(`*s*`, `*len*`)`

Возвращает последовательность длиной *len*, содержащую элементы последовательности *s*, выбранные случайным образом. Элементы в получившейся последовательности располагаются в том порядке, в каком они были отобраны.

`shuffle(`*x* `[,`*random*`])`

Случайным образом перемешивает элементы в списке *x*. В необязательном аргументе *random* передается функция генератора случайных чисел. Эта функция не должна принимать аргументов и должна возвращать число с плавающей точкой в диапазоне `[0.0, 1.0)`.

Распределения случайных вещественных чисел

Следующие функции генерируют случайные вещественные числа. Имена параметров и названия законов распределения соответствуют стандартным названиям, используемым в теории вероятностей и в математической статистике. За дополнительными подробностями вам следует обратиться к соответствующей литературе.

`random()`

Возвращает случайное число в диапазоне `[0.0, 1.0)`.

`uniform(a,b)`

Равномерное распределение. Возвращает числа в диапазоне `[a, b)`.

`betavariate(alpha, beta)`

Бета-распределение. Возвращает числа в диапазоне от 0 до 1. `alpha > -1` и `beta > -1`.

`cunifvariate(mean, arc)`

Круговое равномерное распределение. В аргументе *mean* передается средний угол, а в аргументе *arc* – ширина диапазона распределения относительно среднего угла. Оба эти значения должны указываться в радианах, в диапазоне от `0` до `pi`. Возвращает значения в диапазоне (*mean - arc*/2, *mean + arc*/2).

`expovariate(lambd)`

Экспоненциальное распределение. В аргументе *lambd* передается значение, полученное делением `1.0` на желаемое среднее значение. Возвращает значения в диапазоне `[0, +Infinity)`.

`gammavariate(alpha, beta)`

Гамма-распределение. `alpha > -1`, `beta > 0`.

`gauss(mu, sigma)`

Гауссово распределение[1] со средним значением *mu* и стандартным отклонением *sigma*. Выполняется немного быстрее, чем функция `normalvariate()`.

`lognormvariate(mu, sigma)`

Логарифмически нормальное распределение, то есть нормальное распределение логарифмов чисел, со средним значением *mu* и стандартным отклонением *sigma*.

`normalvariate(mu, sigma)`

Нормальное распределение со средним значением *mu* и стандартным отклонением *sigma*.

`paretovariate(alpha)`

Распределение Парето с параметром формы *alpha*.

---

[1] Часто называют нормальным распределением. – *Прим. перев.*

`triangular([low [, high [, mode]]])`

Треугольное распределение на отрезке `[low, high)` с модальным значением *mode*. По умолчанию аргумент *low* имеет значение `0`, *high* – значение `1.0`, а *mode* – значение посередине между *low* и *high*.

`vonmisesvariate(mu, kappa)`

Распределение фон Мизеса,[1] где *mu* – средний угол в радианах между `0` и `2 * pi`, а *kappa* – неотрицательный коэффициент концентрации. Если аргумент *kappa* имеет нулевое значение, распределение фон Мизеса вырождается до равномерного распределения угла в диапазоне от `0` до `2 * pi`.

`weibullvariate(alpha, beta)`

Распределение Вейбулла с параметром масштабирования *alpha* и параметром формы *beta*.

Примечания

- Функции в этом модуле не предназначены для работы в многопоточных приложениях. В случае необходимости генерировать случайные числа в различных потоках выполнения следует использовать механизм блокировок, чтобы предотвратить одновременный доступ.
- Период генератора случайных чисел (длина последовательности чисел, прежде чем она начнет повторяться) составляет `2**19937-1`.
- Случайные числа, генерируемые этим модулем, можно предсказать, и потому они не должны использоваться в криптографических алгоритмах.
- Имеется возможность создавать новые генераторы случайных чисел за счет создания производных классов от `random.Random` и реализации методов `random()`, `seed()`, `getstate()`, `setstate()` и `jumpahead()`. Все остальные функции в этом модуле в действительности опираются на использование методов класса `Random`. То есть к ним можно обращаться как к методам экземпляра нового генератора случайных чисел.
- Модуль предоставляет два альтернативных класса генераторов случайных чисел – `WichmannHill` и `SystemRandom`. Чтобы использовать их, достаточно создать экземпляр соответствующего класса и вызывать описанные выше функции, как методы. Класс `WichmannHill` реализует генератор Уичмана-Хилла (Wichmann-Hill), который применялся в ранних версиях Python. Класс `SystemRandom` генерирует случайные числа, используя системный генератор случайных чисел `os.urandom()`.

[1] Иначе называемое круговым нормальным рассеиванием. – *Прим. ред.*

15

Структуры данных, алгоритмы и упрощение программного кода

Модули, о которых рассказывается в этой главе, используются для решения типичных задач программирования и имеют отношение к определению структур данных, реализации алгоритмов и позволяют упростить программный код за счет использования итераций, функционального программирования, менеджеров контекста и классов. Эти модули следует рассматривать, как расширения встроенных типов и функций языка Python. Во многих случаях внутренняя реализация обладает чрезвычайно высокой эффективностью и лучше подходит для решения различных проблем, чем встроенные типы и функции.

Модуль abc

Модуль `abc` объявляет метакласс и пару декораторов, с помощью которых определяются новые абстрактные базовые классы.

`ABCMeta`

Метакласс, являющийся представлением абстрактного базового класса. Чтобы определить абстрактный базовый класс, необходимо объявить класс, использующий метакласс `ABCMeta`. Например:

```
import abc
class Stackable:                # В Python 3 используется синтаксис
    __metaclass__ = abc.ABCMeta # class Stackable(metaclass=abc.ABCMETA)
    ...
```

Классы, созданные таким способом, имеют следующие важные отличия от обычных классов:

- Во-первых, если в абстрактном классе объявляются методы или свойства с использованием декораторов `@abstractmethod` и `@abstractproperty`, описываемых ниже, то экземпляры производных классов не могут быть созданы, если эти классы не имеют неабстрактных реализаций этих методов и свойств.

- Во-вторых, абстрактные классы обладают методом класса `register(subclass)`, который может использоваться для регистрации дополнительных типов как логических подклассов. Для любого подкласса, зарегистрированного с помощью этого метода, вызов `isinstance(x, AbstractClass)` будет возвращать `True`, если `x` является экземпляром этого подкласса.
- Наконец, абстрактные классы могут объявлять специальный метод класса `__subclasshook__(cls, subclass)`. Этот метод должен возвращать `True`, если тип класса `subclass` может считаться подклассом класса `cls`, `False` – если класс `subclass` не является подклассом `cls`, или возбуждать исключение `NotImplemented`, если невозможно сделать никаких предположений о принадлежности класса `subclass` к иерархии.

`abstractmethod(method)`

Декоратор, который объявляет метод `method` абстрактным. Когда этот декоратор используется в абстрактном базовом классе, классы, непосредственно объявленные как производные этого базового класса, могут использоваться для создания экземпляров, только если они включают неабстрактное определение метода `method`. Этот декоратор не оказывает влияния на подклассы, зарегистрированные с помощью метода `register()` абстрактного базового класса.

`abstractproperty(fget [, fset [, fdel [, doc]]])`

Создает абстрактное свойство. Этот декоратор принимает те же аргументы, что и обычная функция `property()`. Когда этот декоратор используется в абстрактном базовом классе, классы, непосредственно объявленные как производные этого базового класса, могут использоваться для создания экземпляров, только если они включают неабстрактное определение свойства.

В следующем фрагменте приводится пример объявления простого абстрактного класса:

```
from abc import ABCMeta, abstractmethod, abstractproperty
class Stackable:            # В Python 3 используется синтаксис
    __metaclass__ = ABCMeta # class Stackable(metaclass=ABCMeta)
    @abstractmethod
    def push(self,item):
        pass
    @abstractmethod
    def pop(self):
        pass
    @abstractproperty
    def size(self):
        pass
```

Ниже приводится пример класса, производного от класса `Stackable`:

```
class Stack(Stackable):
    def __init__(self):
        self.items = []
    def push(self,item):
        self.items.append(item)
    def pop(self):
        return self.items.pop()
```

Ниже показано сообщение об ошибке, которое выводится при попытке создать экземпляр класса `Stack`:

```
>>> s = Stack()
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
TypeError: Can't instantiate abstract class Stack with abstract methods size
(Перевод:
Трассировочная информация (самый последний вызов - самый нижний):
  Файл "<stdin>", строка 1, в <модуль>
TypeError: Невозможно создать экземпляр абстрактного класса Stack с абстрактным
методом size
)
>>>
```

Чтобы исправить эту ошибку, необходимо добавить определение свойства `size()` в класс `Stack`. Сделать это можно, либо изменив объявление самого класса `Stack`, либо объявив производный класс и добавив в него необходимые объявления свойств и методов:

```
class CompleteStack(Stack):
    @property
    def size(self):
        return len(self.items)
```

Ниже приводится пример использования законченного объекта стека:

```
>>> s = CompleteStack()
>>> s.push("foo")
>>> s.size
1
>>>
```

См. также

Глава 7 «Классы и объектно-ориентированное программирование», описание модуля numbers в главе 14 (стр. 321) и описание модуля collections ниже, в этой главе (стр. 332).

Модуль array

В модуле `array` объявляется новый тип объектов – `array`, который действует практически так же, как список, но все его элементы могут принадлежать только одному какому-нибудь типу. Тип элементов массива определяется в момент создания с помощью одного из кодов, перечисленных в табл. 15.1.

Таблица 15.1. Коды типов

Код типа	Описание	Тип в языке C	Минимальный размер (в байтах)
'c'	8-битный символ	char	1
'b'	8-битное целое	signed char	1

Код типа	Описание	Тип в языке C	Минимальный размер (в байтах)
'B'	8-битное целое без знака	unsigned char	1
'u'	Символ Юникода	PY_UNICODE	2 или 4
'h'	16-битное целое	short	2
'H'	16-битное целое без знака	unsigned short	2
'i'	Целое	int	4 или 8
'I'	Целое без знака	unsigned int	4 или 8
'l'	Длинное целое	long	4 или 8
'L'	Длинное целое без знака	unsigned long	4 или 8
'f'	Число с плавающей точкой одинарной точности	float	4
'd'	Число с плавающей точкой двойной точности	double	8

Формат представления целых и длинных целых чисел определяется аппаратной архитектурой (это могут быть 32- или 64-битные значения). Когда тип элементов массива определяется кодом 'L' или 'I', в языке Python 2 они возвращаются, как длинные целые числа.

В модуле определяется следующий тип данных:

array(*typecode* [, *initializer*])

Создает массив элементов типа *typecode*. В необязательном аргументе *initializer* передается строка или список значений, используемых для инициализации значений элементов массива. Объект *a* типа array имеет следующие атрибуты и методы:

Атрибут или метод	Описание
a.typecode	Символ с кодом типа, который использовался при создании массива.
a.itemsize	Размер элемента массива (в байтах).
a.append(*x*)	Добавляет в конец массива новый элемент со значением *x*.
a.buffer_info()	Возвращает кортеж (*address*, *length*), содержащий адрес и размер буфера в памяти, выделенного для хранения массива.
a.byteswap()	Меняет порядок следования байтов во всех элементах массива с прямого на обратный, или наоборот. Поддерживается только для целочисленных значений.

(продолжение)

Атрибут или метод	Описание
a.count(*x*)	Возвращает количество вхождений *x* в массиве *a*.
a.extend(*b*)	Добавляет объект *b* в конец массива *a*. Объект *b* *может быть массивом или другим итерируемым объектом, тип элементов которого совпадает с типом элементов массива* *a*.
a.fromfile(*f*, *n*)	Читает *n* элементов (в двоичном формате) из объекта файла *f* и добавляет их в конец массива. Аргумент *f* должен быть объектом файла. Если количество элементов, которое удалось прочитать из файла, окажется меньше, чем *n*, возбуждает исключение EOFError.
a.fromlist(*list*)	Добавляет элементы из списка *list* в конец массива *a*. Аргумент *list* может быть итерируемым объектом.
a.fromstring(*s*)	Добавляет элементы из строки *s* в конец массива, где *s* интерпретируется как строка двоичных значений, как если бы чтение выполнялось методом fromfile().
a.index(*x*)	Возвращает индекс первого вхождения значения *x* в массиве *a*. В случае отсутствия значения *x* возбуждает исключение ValueError.
a.insert(*i*, *x*)	Вставляет новый элемент со значением *x* перед элементом с индексом *i*.
a.pop([*i*])	Удаляет из массива элемент с индексом *i* и возвращает его. Если аргумент *i* опущен, удаляет последний элемент.
a.remove(*x*)	Удаляет первое вхождение *x* из массива. В случае отсутствия значения *x* возбуждает исключение ValueError.
a.reverse()	Переставляет элементы массива в обратном порядке.
a.tofile(*f*)	Записывает все элементы массива в файл *f*. Данные сохраняются в двоичном формате.
a.tolist()	Преобразует массив в обычный список.
a.tostring()	Преобразует массив в строку двоичных данных, как если бы запись выполнялась методом tofile().
a.tounicode()	Преобразует массив в строку Юникода. Возбуждает исключение ValueError, если элементы массива имеют тип, отличный от 'u'.

При попытке вставить в массив элементы, тип которых не совпадает с типом, использовавшимся при создании массива, возбуждается исключение TypeError.

Модуль array может пригодиться, когда потребуется реализовать списки данных с эффективным расходованием памяти и известно, что все элементы списка принадлежат одному типу. Например, чтобы сохранить список из 10 миллионов целых чисел, потребуется примерно 160 Мбайт памяти, тогда как массив из 10 миллионов целых чисел займет всего 40 Мбайт. Не-

смотря на такую экономию памяти, ни одна из основных операций над массивами не выполняется быстрее, чем аналогичные операции над списками, более того, они могут оказаться даже медленнее.

При выполнении вычислений с участием массивов следует быть осторожными с операциями, которые создают списки. Например, когда генераторы списков применяются к массивам, они преобразуют их в списки целиком, уничтожая все преимущества в экономии памяти. В подобных ситуациях создавать новые массивы лучше с помощью выражений-генераторов. Например:

```
a = array.array("i", [1,2,3,4,5])
b = array.array(a.typecode, (2*x for x in a)) # Создаст новый массив из a
```

Поскольку выигрыш от использования массивов заключается в экономии памяти, возможно, окажется более осмысленным выполнить операции непосредственно в исходном массиве. Функция `enumerate()` обеспечивает эффективный способ реализации таких операций, например:

```
a = array.array("i", [1,2,3,4,5])
for i, x in enumerate(a):
    a[i] = 2*x
```

В случае больших массивов операции внутри массива выполняются примерно на 15 процентов быстрее, чем аналогичный программный код, который создает новый массив с помощью выражения-генератора.

Примечания

Массивы, создаваемые с помощью этого модуля, не поддерживают матричные или векторные операции. Например, оператор добавления не добавляет соответствующие элементы массивов – он просто добавит один массив в конец другого. Создавать массивы, позволяющие экономить память и обеспечивающие высокую эффективность операций, можно с помощью расширения `numpy`, доступного по адресу *http://numpy.sourceforge.net/*. Обратите внимание, что это расширение имеет совершенно иной прикладной интерфейс.

Для добавления содержимого одного массива в конец другого можно использовать оператор `+=`. Создавать дубликаты содержимого можно с помощью оператора `*=`.

См. также

Описание модуля struct (стр. 366).

Модуль bisect

Модуль `bisect` обеспечивает возможность поддержания списков в отсортированном состоянии. В своей работе он опирается в основном на использование алгоритма двоичного поиска.

`bisect(`*`list`*`, `*`item`*` [, `*`low`*` [, `*`high`*`]])`

Возвращает индекс в списке *list*, куда следует вставить значение *item*, чтобы сохранить этот список в отсортированном порядке. Аргументы *low* и *high* определяют индексы начала и конца области поиска в списке. Если значение *items* уже присутствует в списке, возвращаемый индекс всегда будет правее существующего элемента списка.

`bisect_left(`*`list`*`, `*`item`*` [, `*`low`*` [, `*`high`*`]])`

Возвращает индекс в списке *list*, куда следует вставить значение *item*, чтобы сохранить этот список в отсортированном порядке. Аргументы *low* и *high* определяют индексы начала и конца области поиска в списке. Если значение *items* уже присутствует в списке, возвращаемый индекс всегда будет левее существующего элемента списка.

`bisect_right(`*`list`*`, `*`item`*` [, `*`low`*` [, `*`high`*`]])`

То же, что и функция `bisect()`.

`insort(`*`list`*`, `*`item`*` [, `*`low`*` [, `*`high`*`]])`

Вставляет элемент *item* в список *list* с учетом порядка сортировки. Если значение *item* уже присутствует в списке, новый элемент вставляется правее его.

`insort_left(`*`list`*`, `*`item`*` [, `*`low`*` [, `*`high`*`]])`

Вставляет элемент *item* в список *list* с учетом порядка сортировки. Если значение *item* уже присутствует в списке, новый элемент вставляется левее его.

`insort_right(`*`list`*`, `*`item`*` [, `*`low`*` [, `*`high`*`]])`

То же, что и `insort()`.

Модуль collections

Модуль `collections` содержит оптимизированные реализации нескольких контейнерных типов, абстрактные базовые классы для различных типов контейнеров и вспомогательные функции для создания именованных кортежей. Все это описывается в следующих разделах.

Типы данных deque и defaultdict

Модуль `collections` объявляет два новых типа контейнеров: `deque` и `defaultdict`.

`deque([`*`iterable`*` [, `*`maxlen`*`]])`

Тип данных, представляющий двустороннюю очередь (название типа `deque` произносится «дек»). В аргументе *iterable* передается итерируемый объект, используемый для заполнения очереди `deque`. *Двусторонняя очередь* позволяет добавлять и удалять элементы из любого конца очереди. Реализация очередей оптимизирована так, что эти операции имеют примерно одинаковую производительность (`O(1)`). Этим очереди выгодно отличаются от списков, где выполнение операций в начале списка может потребовать выпол-

нить сдвиг всех элементов, расположенных правее элемента, над которым выполняется операция. Если передается необязательный аргумент *maxlen*, возвращаемый объект deque превращается в кольцевой буфер указанного размера. То есть при добавлении нового элемента в очередь, в которой уже не осталось свободного места, производится удаление элемента с противоположного конца, чтобы освободить место.

Экземпляр *d* типа deque имеет следующие методы:

d.append(*x*)

Добавляет объект *x* с правой стороны очереди *d*.

d.appendleft(*x*)

Добавляет объект *x* с левой стороны очереди *d*.

d.clear()

Удаляет все элементы из очереди *d*.

d.extend(*iterable*)

Расширяет очередь *d*, добавляя с правой стороны все элементы из итерируемого объекта *iterable*.

d.extendleft(*iterable*)

Расширяет очередь *d*, добавляя с левой стороны все элементы из итерируемого объекта *iterable*. Из-за того, что добавление производится последовательно, по одному элементу, элементы итерируемого объекта *iterable* будут добавлены в очередь *d* в обратном порядке.

d.pop()

Удаляет и возвращает элемент с правой стороны очереди *d*. Если очередь пуста, возбуждает исключение IndexError.

d.popleft()

Удаляет и возвращает элемент с левой стороны очереди *d*. Если очередь пуста, возбуждает исключение IndexError.

d.remove(*item*)

Удаляет первое вхождение элемента *item*. Возбуждает исключение ValueError, если указанное значение не будет найдено.

d.rotate(*n*)

Прокручивает все элементы на *n* позиций вправо. Если в аргументе *n* передается отрицательное значение, прокручивание выполняется влево.

Двусторонние очереди часто незаслуженно забываются многими программистами. Однако этот тип данных предлагает множество преимуществ. Во-первых, очереди этого типа отличаются весьма эффективной реализацией, даже на уровне использования внутренних структур данных, обеспечивающих оптимальное использование кэша процессора. Добавление новых элементов в конец выполняется немногим медленнее, чем во встроенных списках, зато добавление в начало выполняется существенно быстрее. Кроме того, операции добавления новых элементов в двусторонние очереди реализованы с учетом возможности их использования в многопоточных

приложениях, что делает этот тип данных привлекательным для реализации очередей. Двусторонние очереди поддерживают также возможность сериализации средствами модуля pickle.

defaultdict([*default_factory*], ...)

Тип данных, который практически в точности повторяет функциональные возможности словарей, за исключением способа обработки обращений к несуществующим ключам. Когда происходит обращение к несуществующему ключу, вызывается функция, которая передается в аргументе *default_factory*. Эта функция должна вернуть значение по умолчанию, которое затем сохраняется как значение указанного ключа. Остальные аргументы функции defaultdict() в точности те же самые, что передаются встроенной функции dict(). Экземпляры типа defaultdict обладают теми же возможностями, что и встроенные словари. Атрибут default_factory содержит функцию, которая передается функции в первом аргументе, и может изменяться в случае необходимости.

Объекты типа defaultdict удобно использовать в качестве словаря для слежения за данными. Например, предположим, что необходимо отслеживать позицию каждого слова в строке s. Ниже показано, насколько просто это можно реализовать с помощью объекта defaultdict:

```
>>> from collections import defaultdict
>>> s = "yeah but no but yeah but no but yeah"
>>> words = s.split()
>>> wordlocations = defaultdict(list)
>>> for n, w in enumerate(words):
...     wordlocations[w].append(n)
...
>>> wordlocations
defaultdict(<type 'list'>, {'yeah':[0, 4, 8],'but': [1, 3, 5, 7],'no': [2, 6]})
>>>
```

В этом примере операция обращения к элементу словаря wordlocations[w] будет «терпеть неудачу», когда слово встречается впервые. Однако вместо исключения KeyError будет вызвана функция list, переданная в аргументе *default_factory*, которая создаст новое значение. Встроенные словари имеют метод setdefault(), который позволяет добиться того же эффекта, но его использование делает программный код менее наглядным, и к тому же он работает медленнее. Например, инструкцию, добавляющую новый элемент в предыдущем примере, можно было бы заменить инструкцией wordlocations.setdefault(w,[]).append(n). Но она не так очевидна и выполняется почти в два раза медленнее, чем пример с использованием объекта defaultdict.

Именованные кортежи

Кортежи часто используются для представления простых структур данных. Например, кортежи можно использовать для представления сетевых адресов: addr = (*hostname*, *port*). Типичный недостаток кортежей состоит в том, что к отдельным элементам приходится обращаться с помощью числовых индексов, например: addr[0] или addr[1]. Это усложняет чтение программного кода и его сопровождение, потому что приходится запоминать

значение всех индексов (в случае больших кортежей эта ситуация ухудшается еще больше).

Модуль `collections` содержит функцию `namedtuple()`, которая используется для создания подклассов кортежей, которые позволяют обращаться к своим элементам, как к атрибутам.

`namedtuple(`*`typename`*`, `*`fieldnames`*` [, `*`verbose`*`])`

Создает подкласс типа `tuple` с именем *`typename`*. В аргументе *`fieldnames`* передается список имен атрибутов в виде строк. Имена в этом списке должны быть допустимыми идентификаторами Python. Они не должны начинаться с символа подчеркивания, а порядок их следования определяет порядок следования элементов кортежа, например `['hostname','port']`. Кроме того, в аргументе *`fieldnames`* допускается передавать строку, такую как `'hostname port'` или `'hostname, port'`. Возвращаемым значением функции является класс, имя которого передается в аргументе *`typename`*. Этот класс можно использовать для создания экземпляров именованных кортежей. Если в аргументе *`verbose`* передается значение `True`, функция выводит определение класса в поток стандартного вывода.

Ниже приводится пример использования этой функции:

```
>>> from collections import namedtuple
>>> NetworkAddress = namedtuple('NetworkAddress',['hostname','port'])
>>> a = NetworkAddress('www.python.org',80)
>>> a.hostname
'www.python.org'
>>> a.port
80
>>> host, port = a
>>> len(a)
2
>>> type(a)
<class '__main__.NetworkAddress'>
>>> isinstance(a, tuple)
True
>>>
```

В этом примере создается именованный кортеж `NetworkAddress`, который ничем не отличается от обычного кортежа, за исключением поддержки возможности обращаться к его элементам как к атрибутам, например: `a.hostname` или `a.port`. Внутренняя реализация этого класса не использует словарь экземпляра и не увеличивает потребление памяти по сравнению с обычными кортежами. Поддерживаются все обычные операции над кортежами.

Именованные кортежи удобно использовать для определения объектов, которые в действительности являются обычными структурами данных. Например, вместо класса, такого как показано ниже:

```
class Stock(object):
    def __init__(self,name,shares,price):
        self.name = name
        self.shares = shares
        self.price = price
```

Можно было бы определить именованный кортеж:

```
import collections
Stock = collections.namedtuple('Stock','name shares price')
```

Обе версии действуют практически одинаково. Например, обе версии обеспечивают доступ к полям с именами `s.name`, `s.shares` и так далее. Однако именованные кортежи эффективнее расходуют память и поддерживают различные операции над кортежами, такие как распаковывание элементов (например, если имеется список именованных кортежей, эти кортежи можно будет распаковывать в цикле `for`, например: `for name, shares, price in stockList`). Недостатком именованных кортежей является более низкая скорость операции получения значений атрибутов в сравнении с классами. Например, обращение к атрибуту `s.shares` выполняется более чем в два раза медленнее, если `s` является экземпляром именованного кортежа.

Именованные кортежи широко используются в стандартной библиотеке языка Python, где их применение обусловлено, отчасти, историческими причинами – во многих библиотечных модулях кортежи изначально использовались в различных функциях для возврата информации о файлах, кадрах стека и другой низкоуровневой информации. Программный код, использующий эти кортежи, не всегда выглядел достаточно элегантным. Поэтому был выполнен переход к использованию именованных кортежей, что позволило сделать программный код более понятным, без нарушения обратной совместимости. Еще одна проблема именованных кортежей заключается в том, что после того, как вы начнете использовать именованный кортеж, количество его полей должно оставаться неизменным (то есть, если добавить в кортеж новое поле, прежний программный код, использующий его, окажется неработоспособным). Разновидности именованных кортежей используются в библиотеке для добавления новых полей к данным, возвращаемым некоторыми функциями. Например, объект может изначально поддерживать прежний интерфейс кортежей, а позднее к нему могут быть добавлены дополнительные значения, доступные в виде именованных атрибутов.

Абстрактные базовые классы

В модуле `collections` определено несколько абстрактных базовых классов. Назначение их состоит в том, чтобы описать программный интерфейс некоторых типов контейнеров, таких как списки, множества и словари. Можно выделить два основных случая использования этих классов. Во-первых, они могут использоваться, как базовые классы для пользовательских объектов, имитирующих функциональность встроенных контейнерных типов. Во-вторых, они могут использоваться для проверки типов. Например, если требуется убедиться, что объект *s* способен действовать как последовательность, можно вызвать функцию `isinstance(s, collections.Sequence)`.

`Container`

Базовый класс всех контейнеров. Определяет единственный абстрактный метод `__contains__()`, реализующий оператор `in`.

`Hashable`

Базовый класс всех объектов, которые могут использоваться в качестве ключей хеш-таблиц. Определяет единственный абстрактный метод `__hash__()`.

`Iterable`

Базовый класс объектов, поддерживающих протокол итераций. Определяет единственный абстрактный метод `__iter__()`.

`Iterator`

Базовый класс итерируемых объектов. Определяет абстрактный метод `next()`, а также наследует класс `Iterable` и предоставляет реализацию по умолчанию метода `__iter__()`, который просто ничего не делает.

`Sized`

Базовый класс контейнеров, которые позволяют определить размер. Определяет абстрактный метод `__len__()`.

`Callable`

Базовый класс объектов, поддерживающих возможность вызова, как функции. Определяет абстрактный метод `__call__()`.

`Sequence`

Базовый класс объектов, которые выглядят как последовательности. Наследует классы `Container`, `Iterable` и `Sized`, а также определяет абстрактные методы `__getitem__()` и `__len__()`. Кроме того, предоставляет реализацию по умолчанию методов `__contains__()`, `__iter__()`, `__reversed__()`, `index()` и `count()`, которые реализованы исключительно посредством методов `__getitem__()` и `__len__()`.

`MutableSequence`

Базовый класс изменяемых последовательностей. Наследует класс `Sequence` и добавляет абстрактные методы `__setitem__()` и `__delitem__()`. Кроме того, предоставляет реализацию по умолчанию методов `append()`, `reverse()`, `extend()`, `pop()`, `remove()` и `__iadd__()`.

`Set`

Базовый класс объектов, которые действуют как множества. Наследует классы `Container`, `Iterable` и `Sized` и определяет абстрактные методы `__len__()`, `__iter__()` и `__contains__()`. Кроме того, предоставляет реализацию по умолчанию операций над множествами `__le__()`, `__lt__()`, `__eq__()`, `__ne__()`, `__gt__()`, `__ge__()`, `__and__()`, `__or__()`, `__xor__()`, `__sub__()` и `isdisjoint()`.

`MutableSet`

Базовый класс изменяемых множеств. Наследует класс `Set` и добавляет абстрактные методы `add()` и `discard()`. Кроме того, предоставляет реализацию по умолчанию методов `clear()`, `pop()`, `remove()`, `__ior__()`, `__iand__()`, `__ixor__()` и `__isub__()`.

`Mapping`

Базовый класс объектов, поддерживающих возможность отображения (словари). Наследует классы `Sized`, `Iterable` и `Container` и определяет абстракт-

ные методы `__getitem__()`, `__len__()` и `__iter__()`. Предоставляет реализацию по умолчанию методов `__contains__()`, `keys()`, `items()`, `values()`, `get()`, `__eq__()` и `__ne__()`.

`MutableMapping`

Базовый класс изменяемых объектов отображений. Наследует класс `Mapping` и добавляет абстрактные методы `__setitem__()` и `__delitem__()`. Кроме того, предоставляет реализацию по умолчанию методов `pop()`, `popitem()`, `clear()`, `update()` и `setdefault()`.

`MappingView`

Базовый класс представлений отображений. Представление отображения – это объект, который позволяет обращаться к элементам объекта отображения как к множествам. Например, представлением ключей является объект, напоминающий множество, который содержит ключи, имеющиеся в отображении. Подробнее об этом рассказывается в приложении А «Python 3».

`KeysView`

Базовый класс представления ключей отображения. Наследует классы `MappingView` и `Set`.

`ItemsView`

Базовый класс представления элементов отображения. Наследует классы `MappingView` и `Set`.

`ValuesView`

Базовый класс представления пар (*key, item*) отображения. Наследует классы `MappingView` и `Set`.

Соответствующие встроенные типы языка Python уже зарегистрированы в этих базовых классах. Кроме того, используя эти базовые классы, можно писать программы, которые выполняют более точную проверку типов. Например:

```
# Получить последний элемент последовательности
if isinstance(x, collections.Sequence):
    last = x[-1]

# Выполнить итерации по объекту, только если известен его размер
if isinstance(x, collections.Iterable) and isinstance(x, collections.Sized):
    for item in x:
        инструкции

# Добавить новый элемент в множество
if isinstance(x, collections.MutableSet):
    x.add(item)
```

См. также

Глава 7 «Классы и объектно-ориентированное программирование».

Модуль contextlib

Модуль `contextlib` предоставляет декораторы и вспомогательные функции для создания менеджеров контекста, используемых совместно с инструкцией `with`.

`contextmanager(`*`func`*`)`

Декоратор, который создает менеджер контекста из функции-генератора *`func`*. Декораторы используются, как показано ниже:

```
@contextmanager
def foo(args):
    statements
    try:
        yield value
    except Exception as e:
        error handling (if any)
    инструкции
```

Когда интерпретатор встречает инструкцию `with foo(`*`args`*`) as` *`value`*, он вызывает функцию-генератор с указанными аргументами, которая выполняется до первой встреченной инструкции `yield`. Значение, возвращенное инструкцией `yield`, помещается в переменную *`value`*. После этого начинается выполнение тела инструкции `with`. По завершении выполнения тела инструкции `with` возобновляется работа функции-генератора. Если внутри тела инструкции `with` возникнет какое-либо исключение, оно будет передано функции-генератору, где может быть обработано. Если ошибка не может быть обработана функцией-генератором, она должна повторно возбудить исключение. Пример использования декоратора можно найти в разделе «Менеджеры контекста и инструкция with» в главе 5.

`nested(`*`mgr1`*`, `*`mgr2`*`, ..., `*`mgrN`*`)`

Функция, которая вызывает несколько менеджеров контекста *`mgr1`*, *`mgr2`* и так далее, в виде единственной операции. Возвращает кортеж, содержащий различные возвращаемые значения инструкций `with`. Инструкция `with nested(`*`m1`*`,`*`m2`*`) as (`*`x`*`,`*`y`*`): `*`инструкции`* – это то же самое, что и инструкция `with` *`m1`* `as` *`x`*`: with` *`m2`* `as` *`y`*`: `*`инструкции`*. Следует заметить, что если во вложенном менеджере контекста будет перехвачено и обработано какое-либо исключение, внешние менеджеры не получат об этом никакой информации.

`closing(`*`object`*`)`

Создает менеджер контекста, который автоматически вызовет метод *`object`*`.close()` по окончании выполнения тела инструкции `with`. Значением, возвращаемым инструкцией `with`, является сам объект *`object`*.

Модуль functools

Модуль `functools` содержит функции и декораторы, которые удобно использовать для создания высокоуровневых функций и декораторов в функциональном программировании.

`partial(`*`function`* `[, *`*`args`* `[, **`*`kwargs`*`]])`

Создает объект типа `partial`, напоминающий функцию, который при вызове обращается к функции *function* и передает ей позиционные аргументы *args*, именованные аргументы *kwargs*, а также любые дополнительные позиционные и именованные аргументы, переданные объекту при его вызове. Дополнительные позиционные аргументы добавляются в конец *args*, а дополнительные именованные аргументы добавляются в словарь *kwargs*, затирая ранее определенные значения с теми же ключами (если таковые имеются). Обычно функция `partial()` используется, когда требуется многократно вызвать одну и ту же функцию, большая часть аргументов которой остаются неизменными. Например:

```
from functools import partial
mybutton = partial(Button, root, fg="black",bg="white",font="times",size="12")
b1 = mybutton(text="Ok")     # Вызовет Button() с аргументами text="Ok"
b2 = mybutton(text="Cancel") # и остальными, которые были переданы функции
b3 = mybutton(text="Restart")# partial() выше
```

Экземпляр *p* объекта, созданного функцией `partial()`, имеет следующие атрибуты:

Атрибут	Описание
p.func	Функция, которая вызывается при вызове объекта *p*.
p.args	Кортеж, содержащий первые позиционные аргументы, которые передаются функции *p*.func при вызове. Дополнительные позиционные аргументы добавляются в конец этого значения.
p.keywords	Словарь, содержащий именованные аргументы, которые передаются функции *p*.func при вызове. Дополнительные именованные аргументы добавляются в этот словарь.

Будьте внимательны при использовании объектов типа `partial` в качестве замены обычным функциям. Результат, возвращаемый объектом, будет отличаться от того, что возвращается обычной функцией. Например, если функцию `partial()` использовать внутри определения класса, полученный объект будет вести себя, как статический метод, а не как метод экземпляра.

`reduce(`*`function`*`, `*`items`* `[, `*`initial`*`])`

Применяет функцию *function* к элементам в итерируемом объекте *items* и возвращает единственное значение. Функция *function* должна принимать два аргумента, и при первом вызове ей передаются два первых элемента из объекта *items*. Затем она будет получать этот результат и все последующие элементы по очереди, пока не будут исчерпаны все элементы в объекте *items*. В необязательном аргументе *initial* передается начальное значение, которое будет использоваться при первом вызове функции *function*, а также в случае пустого объекта *items*. Эта функция является полным аналогом функции `reduce()`, которая в Python 2 была встроенной функцией. Для сохранения совместимости с будущими версиями интерпретатора используйте функцию из модуля `functools`.

`update_wrapper(`*`wrapper`*`, `*`wrapped`*` [, `*`assigned`*` [, `*`updated`*`]])`

Эту вспомогательную функцию удобно использовать при создании декораторов. Она копирует атрибуты функции *wrapped* в функцию-обертку *wrapper*, делая ее похожей на оригинальную функцию. В аргументе *assigned* передается кортеж с именами атрибутов для копирования, который по умолчанию имеет значение (`'__name__','__module__','__doc__'`). В аргументе *updated* передается кортеж с именами атрибутов функции, являющихся словарями, значения из которых должны быть добавлены в функцию-обертку. По умолчанию является кортежем (`'__dict__',`).

`wraps(`*`function`*` [, `*`assigned`*` [, `*`updated`*` ]])`

Декоратор, который выполняет ту же задачу, что и декоратор `update_wrapper()`. Аргументы *assigned* и *updated* имеют такое же назначение. Обычно этот декоратор используется при создании других декораторов. Например:

```
from functools import wraps
def debug(func):
    @wraps(func)
    def wrapped(*args,**kwargs):
        print("Вызывается %s" % func.__name__)
        r = func(*args,**kwargs)
        print("Выполнен вызов %s" % func.__name__)
    return wrapped

@debug
def add(x,y):
    return x+y
```

См. также

Глава 6 «Функции и функциональное программирование».

Модуль heapq

Модуль `heapq` реализует очереди с приоритетами, используя алгоритмы работы с «кучами». Куча – это простой список упорядоченных элементов, для которого выполняются условия, свойственные куче. В частности, `heap[n] <= heap[2*n+1]` и `heap[n] <= heap[2*n+2]` для всех `n`, начиная с `n = 0`. Элемент `heap[0]` всегда содержит наименьший элемент.

`heapify(`*`x`*`)`

Преобразует список *x* в кучу.

`heappop(`*`heap`*`)`

Удаляет и возвращает наименьший элемент из кучи *heap* с соблюдением условий, предъявляемых к кучам. Возбуждает исключение `IndexError`, если куча *heap* не содержит элементов.

`heappush(`*`heap`*`, `*`item`*`)`

Добавляет элемент *item* в кучу с соблюдением условий, предъявляемых к кучам.

`heappushpop(`*heap*, *item*`)`

Добавляет элемент *item* в кучу и одновременно удаляет наименьший элемент кучи. Это более эффективный способ, чем последовательность вызовов `heappush()` и `heappop()`.

`heapreplace(`*heap*, *item*`)`

Удаляет и возвращает наименьший элемент из кучи *heap*. Одновременно добавляет новый элемент *item*. Операция выполняется с соблюдением условий, предъявляемых к кучам. Это более эффективный способ, чем последовательность вызовов `heappop()` и `heappush()`. Кроме того, возвращаемое значение извлекается из кучи до того, как будет добавлен новый элемент. Поэтому возвращаемое значение может оказаться больше значения *item*. Возбуждает исключение `IndexError`, если куча *heap* не содержит элементов.

`merge(`*s1*, *s2*, ...`)`

Создает итератор, объединяющий отсортированные итерируемые объекты *s1*, *s2* и так далее в одну отсортированную последовательность. Эта функция не выполняет обход объектов, полученных в виде аргументов, а возвращает итератор, который выполняет последовательную обработку данных.

`nlargest(`*n*, *iterable* [, *key*]`)`

Создает список, содержащий *n* наибольших элементов в итерируемом объекте *items*. Первым в возвращаемом списке будет стоять наибольший элемент. В необязательном аргументе *key* передается функция, принимающая единственный аргумент и вычисляющая ключ для каждого элемента в объекте *iterable*, который будет использоваться в операции сравнивания.

`nsmallest(`*n*, *iterable* [, *key*]`)`

Создает список, содержащий *n* наименьших элементов в итерируемом объекте *items*. Первым в возвращаемом списке будет стоять наименьший элемент. В необязательном аргументе *key* передается функция, принимающая единственный аргумент и вычисляющая ключ.

Примечание

Теоретическое обоснование и примеры реализации очередей с приоритетами можно найти во многих книгах, посвященных алгоритмам.

Модуль itertools

Модуль `itertools` содержит функции, создающие эффективные итераторы, которые позволяют выполнять итерации по данным различными способами. Все функции в этом модуле возвращают итераторы, которые можно использовать в инструкции `for` и в других функциях, где применяются итераторы, например в функциях-генераторах и в выражениях-генераторах.

`chain(`*iter1*, *iter2*, ..., *iterN*`)`

На основе группы итераторов (*iter1*,... , *iterN*) создает новый итератор, объединяющий все итераторы воедино. Возвращаемый итератор воспроиз-

водит элементы из объекта *iter1* до их исчерпания. Затем воспроизводятся элементы из объекта *iter2*. Этот процесс продолжается, пока не будут исчерпаны все элементы в *iterN*.

`chain.from_iterable(`*`iterables`*`)`

Альтернативный конструктор создания цепочки итерируемых объектов, где в аргументе *iterables* передается итерируемый объект, возвращающий последовательность итерируемых объектов. Эта операция возвращает тот же результат, который дает следующий фрагмент генератора:

```
for it in iterables:
    for x in it:
        yield x
```

`combinations(`*`iterable`*`, `*`r`*`)`

Создает итератор, который возвращает все возможные последовательности из *r* элементов, взятых из итерируемого объекта *iterable*. Элементы в возвращаемых последовательностях располагаются в том же порядке, в каком они встречаются в исходном объекте *iterable*. Например, если в аргументе *iterable* передать список `[1,2,3,4]`, то вызов `combinations([1,2,3,4], 2)` воспроизведет последовательность `[1,2]`, `[1,3]`, `[1,4]`, `[2,3]`, `[3,4]`.

`count([`*`n`*`])`

Создает итератор, который воспроизводит упорядоченную и непрерывную последовательность целых чисел, начиная с *n*. Если аргумент *n* опущен, в качестве первого значения возвращается число `0`. (Обратите внимание, что этот итератор не поддерживает длинные целые числа. По достижении значения `sys.maxint` счетчик переполнится и итератор продолжит воспроизводить значения, начиная с `-sys.maxint - 1`.)

`cycle(`*`iterable`*`)`

Создает итератор, который в цикле многократно выполняет обход элементов в объекте *iterable*. За кулисами создает копию элементов в объекте *iterable*. Эта копия затем используется для многократного обхода элементов в цикле.

`dropwhile(`*`predicate`*`, `*`iterable`*`)`

Создает итератор, который отклоняет элементы из объекта *iterable*, пока функция *predicate*(*item*) возвращает значение `True`. Как только функция *predicate* вернет `False`, итератор воспроизведет этот элемент и все последующие элементы в итерируемом объекте *iterable*.

`groupby(`*`iterable`*` [, `*`key`*`])`

Создает итератор, который группирует одинаковые элементы из итерируемого объекта *iterable*, следующие друг за другом. Процесс группировки основан на поиске одинаковых элементов. Например, если итерируемый объект *iterable* возвращает один и тот же элемент несколько раз подряд, этот элемент образует группу. Если функция применяется к отсортированному списку, она образует группы по числу уникальных элементов в списке. В необязательном аргументе *key* может передаваться функция, которая будет применяться к каждому элементу; в этом случае в сравнива-

нии соседних элементов участвуют возвращаемые значения этой функции, а не значения самих элементов. Итератор, возвращаемый функцией, воспроизводит кортежи (*key*, *group*), где элемент *key* – это значение ключа для группы, а элемент *group* – это итератор, который возвращает все элементы, попавшие в группу.

`ifilter(`*predicate*, *iterable*`)`

Создает итератор, который воспроизводит только те элементы из объекта *iterable*, для которых функция *predicate*(*item*) возвращает `True`. Если в аргументе *predicate* передать `None`, все элементы в объекте *iterable* будут оцениваться как `True` и будут возвращаться итератором.

`ifilterfalse(`*predicate*, *iterable*`)`

Создает итератор, который воспроизводит только те элементы из объекта *iterable*, для которых функция *predicate*(*item*) возвращает `False`. Если в аргументе *predicate* передать `None`, все элементы в объекте *iterable* будут оцениваться как `False` и будут возвращаться итератором.

`imap(`*function*, *iter1*, *iter2*, ..., *iterN*`)`

Создает итератор, который воспроизводит элементы *function*(*i1*,*i2*, ... *iN*), где *i1*, *i2*,..., *iN* – это элементы, полученные из итераторов *iter1*, *iter2*, ..., *iterN* соответственно. Если в аргументе *function* передать `None`, функция `imap()` вернет кортежи вида (*i1*, *i2*, ..., *iN*). Итерации прекращаются, когда один из указанных итераторов прекращает воспроизводить значения.

`islice(`*iterable*, [*start*,] *stop* [, *step*]`)`

Создает итератор, воспроизводящий элементы, которые вернула бы операция извлечения среза *iterable*[*start*:*stop*:*step*]. Первые *start* элементов пропускаются и итерации прекращаются по достижении позиции, указанной в аргументе *stop*. В необязательном аргументе *step* передается шаг выборки элементов. В отличие от срезов, в аргументах *start*, *stop* и *step* не допускается использовать отрицательные значения. Если аргумент *start* опущен, итерации начинаются с `0`. Если аргумент *step* опущен, по умолчанию используется шаг 1.

`izip(`*iter1*, *iter2*, ... *iterN*`)`

Создает итератор, который воспроизводит кортежи (*i1*, *i2*, ..., *iN*), где значения *i1*, *i2*, ..., *iN* извлекаются из итераторов *iter1*, *iter2*, ..., *iterN* соответственно. Итерации останавливаются, когда какой-либо из исходных итераторов прекращает возвращать значения. Итератор, возвращаемый этой функцией, воспроизводит те же значения, что и встроенная функция `zip()`.

`izip_longest(`*iter1*, *iter2*, ..., *iterN* [,fillvalue=None]`)`

То же, что и функция `izip()`, за исключением того, что возвращаемый итератор продолжает итерации, пока не будут исчерпаны все значения, воспроизводимые итераторами *iter1*, *iter2* и так далее. В качестве недостающих значений для итераторов, которые оказались исчерпаны раньше всех, используется `None`, если не было указано иное значение в именованном аргументе `fillvalue`.

`permutations(`*`iterable`* `[,` *`r`*`])`

Создает итератор, который возвращает все возможные перестановки из *r* элементов, взятых из итерируемого объекта *iterable*. Если аргумент *r* опущен, перестановки будут содержать то же количество элементов, что и в исходном объекте *iterable*.

`product(`*`iter1`*`,` *`iter2`*`, ...` *`iterN`*`,` `[repeat=1])`

Создает итератор, который воспроизводит кортежи, представляющие декартово произведение элементов из *iter1*, *iter2* и так далее. Необязательный именованный аргумент `repeat` определяет количество повторений воспроизведенной последовательности.

`repeat(`*`object`* `[,` *`times`*`])`

Создает итератор, который многократно воспроизводит объект *object*. В необязательном аргументе *times* передается количество повторений. Если этот аргумент не задан, количество повторений будет бесконечным.

`starmap(`*`func`*`,` *`iterable`*`)`

Создает итератор, который воспроизводит значения *func*(**item*), где значение *item* извлекается из итерируемого объекта *iterable*. Может действовать, только если объект *iterable* возвращает элементы, пригодные для использования в вызове функции таким способом.

`takewhile(`*`predicate`*`,` *`iterable`*`)`

Создает итератор, который воспроизводит элементы из итерируемого объекта *iterable*, пока функция *predicate*(*item*) возвращает `True`. Итерации прекращаются, как только функция *predicate* вернет `False`.

`tee(`*`iterable`* `[,` *`n`*`])`

Создает *n* независимых итераторов из итерируемого объекта *iterable*. Созданные итераторы возвращаются в виде кортежа из *n* элементов. По умолчанию аргумент *n* имеет значение 2. Эта функция может принимать любые итерируемые объекты. При этом, когда оригинальный итератор клонируется, в кэше сохраняется его копия, которая используется во всех далее создаваемых итераторах. Будьте очень внимательны и не используйте оригинальный итератор *iterable* после вызова функции `tee()`. В противном случае механизм кэширования будет работать некорректно.

Примеры

Следующие примеры демонстрируют, как действуют некоторые функции из модуля `itertools`:

```
from itertools import *
# Выполнить обход последовательности чисел 0,1,...,10,9,8,...,1
# в бесконечном цикле
for i in cycle(chain(range(10),range(10,0,-1))):
    print i

# Создать список уникальных элементов в объекте a
a = [1,4,5,4,9,1,2,3,4,5,1]
```

```
a.sort()
b = [k for k,g in groupby(a)]   # b = [1,2,3,4,5,9]

# Выполнить обход всех возможных пар значений из объектов x и y
x = [1,2,3,4,5]
y = [10,11,12]
for r in product(x,y):
    print(r)
# Выведет (1,10),(1,11),(1,12), ... (5,10),(5,11),(5,12)
```

Модуль operator

Модуль `operator` содержит функции, обеспечивающие доступ к встроенным операторам и специальным методам интерпретатора, которые описываются в главе 3 «Типы данных и объекты». Например, вызов функции `add(3, 4)` — это аналог операции `3 + 4`. Для операций, имеющих версии, которые изменяют сами операнды, можно использовать такие функции как `iadd(x,y)`, которая является аналогом операции `x += y`. В следующем списке перечислены функции, содержащиеся в модуле `operator`, и их аналоги среди операторов:

Функция	Описание
`add(a, b)`	Возвращает результат выражения `a + b` для чисел
`sub(a, b)`	Возвращает результат выражения `a - b`
`mul(a, b)`	Возвращает результат выражения `a * b` для чисел
`div(a, b)`	Возвращает результат выражения `a / b` (старая версия)
`floordiv(a, b)`	Возвращает результат выражения `a // b`
`truediv(a, b)`	Возвращает результат выражения `a / b` (новая версия)
`mod(a, b)`	Возвращает результат выражения `a % b`
`neg(a)`	Возвращает `-a`
`pos(a)`	Возвращает `+a`
`abs(a)`	Возвращает абсолютное значение `a`
`inv(a), invert(a)`	Возвращает инвертированное значение `a`
`lshift(a, b)`	Возвращает результат выражения `a << b`
`rshift(a, b)`	Возвращает результат выражения `a >> b`
`and_(a, b)`	Возвращает результат выражения `a & b` (битовое И)
`or_(a, b)`	Возвращает результат выражения `a | b` (битовое ИЛИ)
`xor(a, b)`	Возвращает результат выражения `a ^ b` (битовое ИСКЛЮЧАЮЩЕЕ ИЛИ)
`not_(a)`	Возвращает результат выражения `not a`
`lt(a, b)`	Возвращает результат проверки `a < b`
`le(a, b)`	Возвращает результат проверки `a <= b`

eq(*a*, *b*)	Возвращает результат проверки *a* == *b*
ne(*a*, *b*)	Возвращает результат проверки *a* != *b*
gt(a, b)	Возвращает результат проверки *a* > *b*
ge(*a*, *b*)	Возвращает результат проверки *a* >= *b*
truth(*a*)	Возвращает True, если значение *a* является истинным, и False — в противном случае
concat(*a*, *b*)	Возвращает результат выражения *a* + *b* для последовательностей
repeat(a, b)	Возвращает результат выражения *a* * *b* для последовательности *a* и целого числа *b*
contains(*a*, *b*)	Возвращает результат выражения *a* in *b*
countOf(*a*, *b*)	Возвращает количество вхождений *b* в *a*
indexOf(*a*, *b*)	Возвращает индекс первого вхождения *b* в *a*
getitem(*a*, *b*)	Возвращает *a*[*b*]
setitem(*a*, *b*, *c*)	Выполняет операцию *a*[*b*] = *c*
delitem(*a*, *b*)	Выполняет операцию del *a*[*b*]
getslice(*a*, *b*, *c*)	Возвращает срез *a*[*b*:*c*]
setslice(a, b, c, v)	Выполняет операцию *a*[*b*:*c*] = *v*
delslice(*a*, *b*, *c*)	Выполняет операцию del *a*[*b*:*c*]
is_(*a*, *b*)	Возвращает результат выражения *a* is *b*
is_not(*a*, *b*)	Возвращает результат выражения *a* is not *b*

На первый взгляд кажется непонятным, кому может прийти в голову использовать эти функции, если операции, которые они выполняют, проще запрограммировать, применяя обычный синтаксис операторов. Тем не менее эти функции могут пригодиться везде, где используются функции обратного вызова, или там, где в противном случае пришлось бы создавать анонимные функции с помощью инструкции lambda. Например, взгляните на результаты тестирования производительности программного кода, использующего функцию functools.reduce():

```
>>> from timeit import timeit
>>> timeit("reduce(operator.add,a)","import operator; a = range(100)")
12.055853843688965
>>> timeit("reduce(lambda x,y: x+y,a)","import operator; a = range(100)")
25.012306928634644
>>>
```

Обратите внимание, что фрагмент, где в качестве функции обратного вызова используется функция operator.add, выполняется более чем в два раза быстрее по сравнению с версией, где используется выражение lambda x,y: x+y.

Кроме того, модуль operator определяет следующие функции, которые служат обертками вокруг операций доступа к атрибутам и элементам, а также вокруг вызовов методов.

attrgetter(*name* [, *name2* [, ... [, *nameN*]]])

Создает вызываемый объект *f*, при вызове которого *f*(*obj*) возвращается значение *obj.name*. Если функции передается более одного аргумента, вызов *f*(*obj*) возвращает кортеж с результатами. Например, если объект *f* был создан вызовом attrgetter('name','shares'), то вызов *f*(*obj*) вернет кортеж (obj.name, obj.shares). Аргументы *name* могут включать оператор точки доступа к атрибутам. Например, если в аргументе *name* передать строку "address.hostname", то вызов *f*(*obj*) вернет значение *obj*.address.hostname.

itemgetter(*item* [, *item2* [, ... [, *itemN*]]])

Создает вызываемый объект *f*, при вызове которого *f*(*obj*) возвращается значение *obj*[*item*]. Если функции передается более одного аргумента, то вызов *f*(*obj*) вернет кортеж (*obj*[*item*], *obj*[*item2*], ..., *obj*[*itemN*]).

methodcaller(*name* [, **args* [, ***kwargs*]])

Создает вызываемый объект *f*, при вызове которого *f*(*obj*) возвращается *obj*.name(**args*, ***kwargs*).

Эти функции могут пригодиться для оптимизации производительности операций, связанных с обращением к функциям обратного вызова, особенно в таких распространенных операциях, как сортировка данных. Например, чтобы отсортировать список кортежей строк по второму столбцу, можно было бы использовать вызов sorted(*rows*, key=lambda r: r[2]) или sorted(*rows*, key=itemgetter(2)). Вторая версия действует намного быстрее, потому что в ней не выполняются лишние операции, связанные с инструкцией lambda.

16

Работа с текстом и строками

В этой главе описываются модули Python, наиболее часто используемые для работы со строками и с текстом. В центре внимания этой главы будут наиболее типичные строковые операции, такие как обработка текста, сопоставление с шаблонами регулярных выражений и форматирование текста.

Модуль codecs

Модуль `codecs` используется для работы с различными кодировками символов, применяемыми при вводе-выводе текста Юникода. Модуль позволяет определять новые кодировки символов и обеспечивает возможность обработки символьных данных с применением широкого спектра существующих кодировок, таких как UTF-8, UTF-16 и других. При разработке приложений намного более типично просто использовать существующие кодировки, поэтому именно эту тему мы и будем обсуждать здесь. Если вам потребуется создавать новые кодировки, за дополнительными подробностями обращайтесь к электронной документации.

Низкоуровневый интерфейс модуля codecs

Каждой кодировке символов присвоено определенное имя, такое как `'utf-8'` или `'big5'`. Следующая функция выполняет поиск кодировки.

`lookup(`*`encoding`*`)`

Отыскивает кодировку в реестре. В аргументе *`encoding`* передается строка, такая как `'utf-8'`. Если искомая кодировка в реестре отсутствует, возбуждается исключение `LookupError`. В противном случае возвращается экземпляр *`c`* класса `CodecInfo`.

Экземпляр *`c`* класса `CodecInfo` обладает следующими методами:

`c``.encode(`*`s`* `[, ` *`errors`*`])`

Функция кодирования, которая кодирует строку *`s`* Юникода и возвращает кортеж (*`bytes, length_consumed`*). Элемент *`bytes`* – **это строка 8-битных сим-**

волов, или массив байтов, содержащая кодированные данные. Элемент *length_consumed* – это число, отражающее количество символов в строке *s*, которые были кодированы. Необязательный аргумент *errors* определяет политику обработки ошибок кодирования и по умолчанию имеет значение 'strict'.

c.decode(*bytes* [, *errors*])

Функция кодирования, которая декодирует строку байтов *bytes* и возвращает кортеж (*s*, *length_consumed*). Элемент *s* – это строка Юникода, а *length_consumed* – количество байтов в аргументе *bytes*, которые были декодированы. Необязательный аргумент *errors* определяет политику обработки ошибок кодирования и по умолчанию имеет значение 'strict'.

c.streamreader(*bytestream* [, *errors*])

Возвращает экземпляр класса StreamReader, который используется для чтения декодированных данных. В аргументе *bytestream* передается объект, поддерживающий интерфейс файлов, открытый в двоичном режиме. Необязательный аргумент *errors* определяет политику обработки ошибок кодирования и по умолчанию имеет значение 'strict'. Экземпляр *r* класса StreamReader поддерживает следующие низкоуровневые операции ввода-вывода:

Метод	Описание
r.read([*size* [, *chars* [, *firstline*]]])	Возвращает последние *chars* символов декодированного текста. Аргумент *size* определяет максимальное число байтов, которые могут быть прочитаны из потока байтов, и используется для управления внутренним механизмом буферизации. В аргументе *firstline* передается флаг. Если он установлен, функция возвращает первую строку текста, даже если далее в файле возникли ошибки декодирования.
r.readline([*size* [, *keepends*]])	Возвращает одну строку декодированного текста. Флаг *keepends* определяет, должны ли сохраняться символы завершения строки (по умолчанию имеет значение *True*).
r.readlines([*size* [, *keepends*]])	Читает все строки в список.
r.reset()	Очищает внутренние буферы и информацию о состоянии объекта.

c.streamwriter(*bytestream* [, *errors*])

Возвращает экземпляр класса StreamWriter, который используется для записи кодированных данных. В аргументе *bytestream* передается объект, поддерживающий интерфейс файлов, открытый в двоичном режиме. Необязательный аргумент *errors* определяет политику обработки ошибок кодирования и по умолчанию имеет значение 'strict'. Экземпляр *w* клас-

са StreamWriter поддерживает следующие низкоуровневые операции ввода-вывода:

Метод	Описание
w.write(*s*)	Выводит кодированное представление строки *s*.
w.writelines(*lines*)	Выводит список строк *lines* в файл.
w.reset()	Очищает внутренние буферы и информацию о состоянии объекта.

c.incrementalencoder([*errors*])

Возвращает экземпляр класса IncrementalEncoder, который может использоваться для кодирования строк в несколько этапов. Необязательный аргумент *errors* по умолчанию имеет значение 'strict'. Экземпляр *e* класса IncrementalEncoder обладает следующими методами:

Метод	Описание
e.encode(*s* [,*final*])	Возвращает кодированное представление строки *s* в виде строки байтов. Аргумент *final* должен устанавливаться в значение True в последнем вызове метода encode().
e.reset()	Очищает внутренние буферы и информацию о состоянии объекта.

c.incrementaldecoder([*errors*])

Возвращает экземпляр класса IncrementalDecoder, который может использоваться для декодирования строк байтов в несколько этапов. Необязательный аргумент *errors* по умолчанию имеет значение 'strict'. Экземпляр *d* класса IncrementalDecoder обладает следующими методами:

Метод	Описание
d.decode(*bytes* [,*final*])	Возвращает декодированную строку, представляющую последовательность кодированных байтов *bytes*. Аргумент *final* должен устанавливаться в значение True в последнем вызове метода decode().
d.reset()	Очищает внутренние буферы и информацию о состоянии объекта.

Функции ввода-вывода

Модуль codecs предоставляет коллекцию высокоуровневых функций, позволяющих упростить реализацию операций ввода-вывода кодированного текста. В большинстве случаев программисты предпочитают использовать эти функции вместо низкоуровневого интерфейса модуля codecs, описанного в первом разделе.

`open(filename, mode[, encoding[, errors[, buffering]]])`

Открывает файл *filename* в режиме *mode* и обеспечивает прозрачное кодирование/декодирование данных в соответствии с кодировкой, указанной в аргументе *encoding*. В аргументе *errors* передается одно из значений: 'strict', 'ignore', 'replace', 'backslashreplace' или 'xmlcharrefreplace'. По умолчанию используется значение 'strict'. Аргумент *buffering* имеет то же назначение, что и одноименный аргумент встроенной функции open(). Независимо от режима, указанного в аргументе *mode*, файл всегда открывается в двоичном режиме. В Python 3 вместо функции codecs.open() можно использовать встроенную функцию open().

`EncodedFile(file, inputenc[, outputenc [, errors]])`

Класс, представляющий собой обертку вокруг существующего объекта файла *file*, выполняющую кодирование. Перед записью в файл данные сначала интерпретируются в соответствии с кодировкой *inputenc*, а затем записываются в файл с использованием кодировки *outputenc*. Декодирование данных, прочитанных из файла, выполняется в соответствии с кодировкой *inputenc*. Если аргумент *outputenc* опущен, он по умолчанию получает значение *inputenc*. Аргумент *errors* имеет то же назначение, что и в функции open(), и по умолчанию имеет значение 'strict'.

`iterencode(iterable, encoding [, errors])`

Функция-генератор, которая последовательно кодирует все строки в объекте *iterable* в соответствии с кодировкой *encoding*. Аргумент *errors* по умолчанию имеет значение 'strict'.

`iterdecode(iterable, encoding [, errors])`

Функция-генератор, которая последовательно декодирует все строки байтов в объекте *iterable* в соответствии с кодировкой *encoding*. Аргумент *errors* по умолчанию имеет значение 'strict'.

Полезные константы

Модуль codecs определяет следующие константы маркеров порядка следования байтов, которые могут использоваться как вспомогательное средство в интерпретации файлов, когда кодировка данных заранее не известна. Иногда эти маркеры записываются в начало файла, чтобы обозначить кодировку символов, и могут использоваться для выбора соответствующей кодировки.

Константа	Описание
BOM	Порядок следования байтов определяется аппаратной архитектурой (BOM_BE или BOM_LE)
BOM_BE	Маркер прямого (big-endian) порядка следования байтов ('\xfe\xff')
BOM_LE	Маркер обратного (little-endian) порядка следования байтов ('\xff\xfe')
BOM_UTF8	Маркер кодировки UTF-8 ('\xef\xbb\xbf')

Константа	Описание
BOM_UTF16_BE	Маркер прямого (big-endian) порядка следования байтов в 16-битовой кодировке UTF-16 (`'\xfe\xff'`)
BOM_UTF16_LE	Маркер обратного (little-endian) порядка следования байтов в 16-битовой кодировке UTF-16 (`'\xfe\xff'`)
BOM_UTF32_BE	Маркер прямого (big-endian) порядка следования байтов в 32-битовой кодировке UTF-32 (`'\x00\x00\xfe\xff'`)
BOM_UTF32_LE	Маркер обратного (little-endian) порядка следования байтов в 32-битовой кодировке UTF-32 (`'\xff\xfe\x00\x00'`)

Стандартные кодировки

Ниже приводится список некоторых из наиболее часто используемых кодировок символов. Имена кодировок приводятся в том виде, в каком они передаются функциям, таким как `open()` или `lookup()`. Полный список можно найти в электронной документации с описанием модуля `codecs` (*http://docs.python.org/library/codecs*).

Название кодировки	Описание
ascii	7-битная кодировка ASCII
cp437	Расширенный набор символов ASCII из MS-DOS
cp1252	Расширенный набор символов ASCII из Windows
latin-1, iso-8859-1	Набор ASCII, дополненный символами из набора Latin
utf-16	UTF-16
utf-16-be	UTF-16 с прямым порядком следования байтов
utf-16-le	UTF-16 с обратным порядком следования байтов
utf-32	UTF-32
utf-32-be	UTF-32 с прямым порядком следования байтов
utf-32-le	UTF-32 с обратным порядком следования байтов
utf-8	UTF-8

Примечания

- Дополнительные сведения об использовании модуля `codecs` приводятся в главе 9 «Ввод и вывод».
- Информацию о том, как создавать новые кодировки символов, можно найти в электронной документации.
- Особое внимание требуется уделять входным аргументам операций `encode()` и `decode()`. Всем операциям `encode()` должны передаваться строки Юникода, а всем операциям `decode()` – строки байтов. В Python 2 это требование соблюдается не так строго, но в Python 3 различия между типами строк имеют существенное значение. Например, в Python 2 име-

ются кодировки, которые отображают строки байтов в строки байтов (например, кодировка «bz2»). Они недоступны в Python 3 и потому не должны использоваться, если вы учитываете задачу совместимости.

Модуль re

Модуль `re` используется для выполнения операций сопоставления с шаблонами регулярных выражений и замены фрагментов строк. Модуль поддерживает операции как со строками Юникода, так и со строками байтов. Шаблоны регулярных выражений определяются как строки, состоящие из смеси текста и последовательностей специальных символов. В шаблонах часто используются специальные символы и символ обратного слэша, поэтому они обычно оформляются, как «сырые» строки, такие как `r'(?P<int>\d+)\.(\d*)'`. В оставшейся части этого раздела все шаблоны регулярных выражений будут записываться с использованием синтаксиса «сырых» строк.

Синтаксис шаблонов

Ниже приводится список последовательностей специальных символов, которые используются в шаблонах регулярных выражений:

Символ(ы)	Описание
text	Соответствует строке *text*.
.	Соответствует любому символу, кроме символа перевода строки.
^	Соответствует позиции начала строки.
$	Соответствует позиции конца строки.
*	Ноль или более повторений предшествующего выражения; соответствует максимально возможному числу повторений.
+	Одно или более повторений предшествующего выражения; соответствует максимально возможному числу повторений.
?	Ноль или одно повторение предшествующего выражения.
*?	Ноль или более повторений предшествующего выражения; соответствует минимально возможному числу повторений.
+?	Одно или более повторений предшествующего выражения; соответствует минимально возможному числу повторений.
??	Ноль или одно повторение предшествующего выражения; соответствует минимально возможному числу повторений.
{m}	Соответствует точно *m* повторениям предшествующего выражения.
{m, n}	Соответствует от *m* до *n* повторений предшествующего выражения. Если аргумент *m* опущен, он принимается равным 0. Если аргумент *n* опущен, он принимается равным бесконечности.

Символ(ы)	Описание
{m, n}?	От *m* до *n* повторений предшествующего выражения; соответствует минимально возможному числу повторений.
[...]	Соответствует любому символу, присутствующему в множестве, таком как r'[abcdef]' или r'[a-zA-z]'. Специальные символы, такие как *, утрачивают свое специальное значение внутри множества.
[^...]	Соответствует любому символу, не присутствующему в множестве, таком как r'[^0-9]'.
A|*B*	Соответствует либо *A*, либо *B*, где *A* и *B* являются регулярными выражениями.
(...)	Подстрока, соответствующая регулярному выражению в круглых скобках, интерпретируется как группа и сохраняется. Содержимое группы может быть получено с помощью метода group() объектов класса MatchObject, которые возвращаются операцией поиска совпадений.
(?aiLmsux)	Символы "a", "i", "L", "m", "s", "u" и "x" интерпретируются как флаги, соответствующие флагам re.A, re.I, re.L, re.M, re.S, re.U, re.X, которые передаются методу re.compile(). Флаг "a" доступен только в Python 3.
(?:...)	Соответствует регулярному выражению в круглых скобках, но совпавшая подстрока не сохраняется.
(?P<*name*>...)	Подстрока, соответствующая регулярному выражению в круглых скобках, интерпретируется как именованная группа с именем *name*. Имя группы должно быть допустимым идентификатором Python.
(?P=*name*...)	Соответствует тому тексту, который ранее уже совпал с именованной группой *name*.
(?#...)	Комментарий. Содержимое в круглых скобках игнорируется.
(?=...)	Соответствует предшествующему выражению, только если вслед за ним обнаруживается совпадение с шаблоном в круглых скобках. Например, выражение r'Hello (?=World)' совпадет с подстрокой 'Hello ', только если за ней будет следовать подстрока 'World'.
(?!...)	Соответствует предшествующему выражению, только если вслед за ним отсутствует совпадение с шаблоном в круглых скобках. Например, выражение r'Hello (?!World)' совпадет с подстрокой 'Hello ' , только если за ней не будет следовать подстрока 'World'.
(?<=...)	Соответствует следующему выражению, только если перед ним обнаруживается совпадение с шаблоном в круглых скобках. Например, выражение r'(?<=abc)def' совпадет с подстрокой 'def', только если ей будет предшествовать подстрока 'abc'.

(продолжение)

Символ(ы)	Описание
(?<!...)	Соответствует следующему выражению, только если перед ним отсутствует совпадение с шаблоном в круглых скобках. Например, выражение r'(?<!abc)def' совпадет с подстрокой 'def', только если ей не будет предшествовать подстрока 'abc'.
(?(*id*|*name*)*ypat*|*npat*)	Проверяет, существует ли группа регулярного выражения с числовым идентификатором *id* или с именем *name*. Если такая группа существует, определяется соответствие регулярному выражению *ypat*. В противном случае определяется соответствие необязательному регулярному выражению *npat*. Например, выражение r'(Hello)?(?(1) World|Howdy)' совпадет со строкой 'Hello World' или со строкой 'Howdy'.

Стандартные экранированные последовательности, такие как '\n' и '\t', точно так же интерпретируются и в регулярных выражениях (например, выражению r'\n+' будет соответствовать один или более символов перевода строки). Кроме того, литералы символов, которые в регулярных выражениях имеют специальное значение, можно указывать, предваряя их символом обратного слэша. Например, выражению r'*' соответствует символ *. Дополнительно ряд экранированных последовательностей, начинающихся символом обратного слэша, соответствуют специальным символам:

Символ(ы)	Описание
число	Соответствует фрагменту текста, совпавшему с группой с указанным номером. Группы нумеруются от 1 до 99, слева направо.
\A	Соответствует только началу строки.
\b	Соответствует пустой строке в позиции начала или конца слова. Под *словом* подразумевается последовательность алфавитно-цифровых символов, завершающаяся пробельным или любым другим не алфавитно-цифровым символом.
\B	Соответствует пустой строке не в позиции начала или конца слова.
\d	Соответствует любой десятичной цифре. То же, что и выражение r'[0-9]'.
\D	Соответствует любому нецифровому символу. То же, что и выражение r'[^0-9]'.
\s	Соответствует любому пробельному символу. То же, что и выражение r'[\t\n\r\f\v]'.
\S	Соответствует любому непробельному символу. То же, что и выражение r'[^\t\n\r\f\v]'.
\w	Соответствует любому алфавитно-цифровому символу.

Символ(ы)	Описание
\W	Соответствует любому символу, не относящемуся к множеству \w.
\Z	Соответствует только концу строки.
\\	Соответствует самому символу обратного слэша.

Специальные символы \d, \D, \s, \S, \w и \W интерпретируются иначе при сопоставлении со строками Юникода. В данном случае они совпадают со всеми символами Юникода, соответствующими описанным свойствам. Например, \d совпадает со всеми символами Юникода, которые классифицируются как цифры, будь то европейские, арабские или индийские цифры, каждые из которых занимают различные диапазоны символов Юникода.

Функции

Ниже перечислены функции, которые используются для выполнения операций сопоставления с шаблоном и замены:

`compile(`*`str`* `[,` *`flags`*`])`

Компилирует строку с шаблоном регулярного выражения и создает объект регулярного выражения. Этот объект может передаваться в аргументе *pattern* всем функциям, описываемым ниже. Данный объект также предоставляет ряд методов, которые описываются чуть ниже. В аргументе *flags* передается битная маска следующих флагов, объединенных битовой операцией ИЛИ:

Флаг	Описание
A или ASCII	Сопоставление выполняется только с 8-битными символами ASCII (только в Python 3).
I или IGNORECASE	Сопоставление выполняется без учета регистра символов.
L или LOCALE	При сопоставлении со специальными символами \w, \W, \b и \B используются региональные настройки.
M или MULTILINE	Обеспечивает совпадение символов ^ и $ с началом и концом каждой строки в тексте, помимо начала и конца самого текста. (Обычно символы ^ и $ совпадают только с началом и концом всего текста.)
S или DOTALL	Обеспечивает совпадение символа точки (.) со всеми символами, включая символ перевода строки.
U или UNICODE	При сопоставлении с символами \w, \W, \b и \B используется информация из базы данных о свойствах символов. (Только в Python 2. В Python 3 Юникод используется по умолчанию.)
X или VERBOSE	Игнорирует неэкранированные пробельные символы и комментарии в строке шаблона.

`escape(`*`string`*`)`

Возвращает строку, в которой все не алфавитно-цифровые символы экранированы символом обратного слэша.

`findall(`*`pattern`*`, `*`string`* `[,`*`flags`*`])`

Возвращает список всех неперекрывающихся совпадений с шаблоном *pattern* в строке *string*, включая пустые совпадения. Если шаблон имеет группы, возвращает список фрагментов текста, совпавших с группами. Если в шаблоне присутствует более одной группы, каждый элемент в списке будет представлен кортежем, содержащим текст из каждой группы. Аргумент *flags* имеет то же назначение, что и в функции `compile()`.

`finditer(`*`pattern`*`, `*`string`*`, [, `*`flags`*`])`

То же, что и функция `findall()`, но вместо списка возвращает итератор. Данный итератор воспроизводит элементы типа `MatchObject`.

`match(`*`pattern`*`, `*`string`* `[, `*`flags`*`])`

Проверяет наличие совпадения с шаблоном *pattern* в начале строки *string*. В случае успеха возвращает объект типа `MatchObject`, в противном случае возвращается `None`. Аргумент *flags* имеет то же назначение, что и в функции `compile()`.

`search(`*`pattern`*`, `*`string`* `[, `*`flags`*`])`

Отыскивает в строке *string* первое совпадение с шаблоном *pattern*. Аргумент *flags* имеет то же назначение, что и в функции `compile()`. В случае успеха возвращает объект типа `MatchObject`; если совпадений не найдено, возвращается `None`.

`split(`*`pattern`*`, `*`string`* `[, `*`maxsplit`* `= 0])`

Разбивает строку *string* по совпадениям с шаблоном *pattern*. Возвращает список строк, включая текст, совпавший с группами, присутствующими в шаблоне. В аргументе *maxsplit* передается максимальное количество выполняемых разбиений. По умолчанию выполняются все возможные разбиения.

`sub(`*`pattern`*`, `*`repl`*`, `*`string`* `[, `*`count`* `= 0])`

Замещает текстом *repl* самые первые неперекрывающиеся совпадения с шаблоном *pattern* в строке *string*. Аргумент *repl* может быть строкой или функцией. Если в этом аргументе передается функция, при вызове ей будет передаваться объект типа `MatchObject`, и она должна возвращать строку замены. Если в аргументе *repl* передается строка, в ней допускается использовать обратные ссылки на группы в шаблоне, такие как '`\6`'. Для ссылки на именованные группы можно использовать последовательность '`\g<`*name*`>`'. Аргумент *count* определяет максимальное количество подстановок. По умолчанию замещаются все найденные совпадения. Эти функции не принимают аргумент *flags*, как функция `compile()`, тем не менее аналогичного эффекта можно добиться, воспользовавшись нотацией `(?iLmsux)`, описанной выше в этом разделе.

`subn(`*`pattern`*`, `*`repl`*`, `*`string`*` [, `*`count`*` = 0])`

То же, что и `sub()`, но возвращает кортеж, содержащий новую строку и количество выполненных подстановок.

Объекты регулярных выражений

Объект скомпилированного регулярного выражения *r*, который создается функцией `compile()`, обладает следующими атрибутами и методами:

r`.flags`

Значение аргумента *flags*, с которым был скомпилирован объект регулярного выражения, или `0`, если ни один из флагов не был указан.

r`.groupindex`

Словарь, отображающий символические имена групп, определяемых как `r'(?P<id>)'`, в порядковые номера групп.

r`.pattern`

Строка шаблона, из которой был скомпилирован объект регулярного выражения.

r`.findall(`*`string`*` [, `*`pos`*` [, `*`endpos`*`]])`

Этот метод идентичен функции `findall()`. В аргументах *pos* и *endpos* передаются начальная и конечная позиция поиска.

r`.finditer(`*`string`*` [, `*`pos`*` [, `*`endpos`*`]])`

Этот метод идентичен функции `finditer()`. В аргументах *pos* и *endpos* передаются начальная и конечная позиции поиска.

r`.match(`*`string`*` [, `*`pos`*`] [, `*`endpos`*`])`

Проверяет наличие совпадения с шаблоном *pattern* в строке *string*. В аргументах *pos* и *endpos* передаются начальная и конечная позиции поиска. В случае успеха возвращает объект типа `MatchObject`, в противном случае возвращается `None`.

r`.search(`*`string`*` [, `*`pos`*`] [, `*`endpos`*`])`

Отыскивает в строке *string* первое совпадение с шаблоном *pattern*. В аргументах *pos* и *endpos* передаются начальная и конечная позиции поиска. В случае успеха возвращает объект типа `MatchObject`; если совпадений не найдено, возвращается `None`.

r`.split(`*`string`*` [, `*`maxsplit`*` = 0])`

Этот метод идентичен функции `split()`.

r`.sub(`*`repl`*`, `*`string`*` [, `*`count`*` = 0])`

Этот метод идентичен функции `sub()`.

r`.subn(`*`repl`*`, `*`string`*` [, `*`count`*` = 0])`

Этот метод идентичен функции `subn()`.

Объекты MatchObject

Экземпляры класса `MatchObject` возвращаются функциями `search()` и `match()` и содержат информацию о группах, а также о позициях найденных совпадений. Экземпляр *m* класса `MatchObject` обладает следующими методами и атрибутами:

m.`expand(`*template*`)`

Возвращает строку, полученную заменой совпавших фрагментов экранированными последовательностями в строке *template*. Обратные ссылки, такие как "\1" и "\2", и именованные ссылки, такие как "\g<n>" и "\g<name>", замещаются содержимым соответствующих групп. Обратите внимание, что эти последовательности должны быть оформлены как «сырые» строки или с дополнительным символом обратного слэша, например: `r'\1'` или `'\\1'`.

m.`group([`*group1*, *group2*, ...`])`

Возвращает одну или более подгрупп в совпадении. Аргументы определяют номера групп или их имена. Если имя группы не задано, возвращается совпадение целиком. Если указана только одна группа, возвращается строка, содержащая текст, совпавший с группой. В противном случае возвращается кортеж, содержащий совпадения со всеми указанными группами. Если было запрошено содержимое группы с недопустимым именем или номером, возбуждает исключение `IndexError`.

m.`groups([`*default*`])`

Возвращает кортеж, содержащий совпадения со всеми группами в шаблоне. В аргументе *default* указывается значение, возвращаемое для групп, не участвовавших в совпадении (по умолчанию имеет значение `None`).

m.`groupdict([`*default*`])`

Возвращает словарь, содержащий совпадения с именованными группами. В аргументе *default* указывается значение, возвращаемое для групп, не участвовавших в сопоставлении (по умолчанию имеет значение `None`).

m.`start([`*group*`])`

m.`end([`*group*`])`

Эти два метода возвращают индексы начала и конца совпадения с группой в строке. Если аргумент *group* опущен, возвращаются позиции всего совпадения. Если группа существует, но не участвовала в сопоставлении, возвращается `None`.

m.`span([`*group*`])`

Возвращает кортеж из двух элементов (*m*.`start(`*group*`)`, *m*.`end(`*group*`)`). Если совпадений с группой *group* не было обнаружено, возвращается кортеж (`None`, `None`). Если аргумент *group* опущен, возвращаются позиции начала и конца всего совпадения.

m.`pos`

Значение аргумента *pos*, переданное методу `search()` или `match()`.

m.endpos

Значение аргумента *endpos*, переданное методу search() или match().

m.lastindex

Числовой индекс последней совпавшей группы. Если ни одна группа не совпала или в шаблоне нет ни одной группы, возвращается None или в шаблоне нет ни одной группы.

m.lastgroup

Имя последней совпавшей группы. Если ни одна группа не совпала или в шаблоне нет ни одной группы, возвращается None.

m.re

Объект регулярного выражения, чьим методом match() или search() был создан данный экземпляр класса MatchObject.

m.string

Строка, переданная методу match() или search().

Пример

Следующий пример демонстрирует способы использования модуля re для поиска, извлечения и замены фрагментов текста в строке.

```
import re
text = "Guido will be out of the office from 12/15/2012 - 1/3/2013."

# Шаблон регулярного выражения для поиска дат
datepat = re.compile('(\d+)/(\d+)/(\d+)')

# Найти и вывести все даты
for m in datepat.finditer(text):
    print(m.group())

# Отыскать все даты и вывести их в другом формате
monthnames = [None,'Jan','Feb','Mar','Apr','May','Jun',
              'Jul','Aug','Sep','Oct','Nov','Dec']
for m in datepat.finditer(text):
    print ("%s %s, %s" % (monthnames[int(m.group(1)], m.group(2), m.group(3)))

# Заменить все даты их значениями в европейском формате (день/месяц/год)
def fix_date(m):
    return "%s/%s/%s" % (m.group(2),m.group(1),m.group(3))
newtext = datepat.sub(fix_date, text)

# Альтернативный способ замены
newtext = datepat.sub(r'\2/\1/\3', text)
```

Примечания

- Подробное описание теории и реализации поддержки регулярных выражений можно найти в учебниках по устройству компиляторов. Кроме

того, полезно будет ознакомиться с книгой «Mastering Regular Expressions» Джеффри Фридла (Jeffrey Friedl) (O'Reilly & Associates, 1997).[1]

- Основные сложности, возникающие при работе с модулем `re`, – это создание регулярных выражений. В их разработке вам может пригодиться такой инструмент, как Kodos (*http://kodos.sourceforget.net*).

Модуль string

Модуль `string` содержит множество констант и функций, которые могут пригодиться для обработки строк. В нем также объявляются классы, предназначенные для реализации новых механизмов форматирования строк.

Константы

Следующие константы определяют различные множества символов, которые могут пригодиться в различных операциях над строками.

Константа	Описание
`ascii_letters`	Строка, содержащая все строчные и заглавные символы ASCII
`ascii_lowercase`	Строка `'abcdefghijklmnopqrstuvwxyz'`
`ascii_uppercase`	Строка `'ABCDEFGHIJKLMNOPQRSTUVWXYZ'`
`digits`	Строка `'0123456789'`
`hexdigits`	Строка `'0123456789abcdefABCDEF'`
`letters`	Конкатенация строк `lowercase` и `uppercase`
`lowercase`	Строка, содержащая все строчные символы, соответствующие текущим региональным настройкам системы
`octdigits`	Строка `'01234567'`
`punctuation`	Строка с символами пунктуации из набора ASCII
`printable`	Строка печатаемых символов – комбинация значений `letters`, `digits`, `punctuation` и `whitespace`
`uppercase`	Строка, содержащая все заглавные символы, соответствующие текущим региональным настройкам системы
`whitespace`	Строка, содержащая все пробельные символы. В их число обычно входят: пробел, табуляция, перевод строки, возврат каретки, перевод формата и вертикальная табуляция

Примечательно, что содержимое некоторых из этих констант (например, `letters` и `uppercase`) зависит от региональных настроек системы.

[1] Фридл Д. «Регулярные выражения. 3-е издание». – Пер. с англ. – СПб.: Символ-Плюс, 2008. – *Прим. перев.*

Объекты типа Formatter

Строковый метод `str.format()` позволяет реализовать дополнительное форматирование строк. Как было показано в главе 3 «Типы данных и объекты» и в главе 4 «Операторы и выражения», этот метод способен обращаться к элементам последовательностей или словарей, к атрибутам объектов и выполнять другие подобные операции. В модуле `string` определен класс `Formatter`, который может использоваться для реализации нестандартных операций форматирования. Этот класс содержит атрибуты и методы, реализующие операции форматирования строк, и позволяет изменять их.

`Formatter()`

Создает новый экземпляр класса `Formatter`. Экземпляр `f` класса `Formatter` поддерживает следующие операции.

f`.format(`*format_string*`, *`*args`*`, **`*kwargs*`)`

Форматирует строку *format_string*. По умолчанию выводит ту же строку, что и метод `format_string.format(`*`*args, **kwargs`*`)`. Например, вызов `f.format("{name} is {0:d} years old", 39,name="Dave")` вернет строку `"Dave is 39 years old"`.

f`.vformat(`*format_string*`, `*args*`, `*kwargs*`)`

Метод, который фактически выполняет работу метода `f.format()`. В аргументе *args* передается кортеж позиционных аргументов, а в аргументе *kwargs* – словарь именованных аргументов. Если аргументы уже сохранены в кортеже и в словаре, можно использовать этот метод, так как он обладает более высокой скоростью работы.

f`.parse(`*format_string*`)`

Создает итератор для парсинга содержимого строки формата *format_string*. Итератор просматривает строку формата и воспроизводит кортежи вида (*literal_text*, *field_name*, *format_spec*, *conversion*). Поле *literal_text* содержит произвольный текст, предшествующий спецификатору формата, заключенному в фигурные скобки { ... }. Может содержать пустую строку, если перед спецификатором отсутствует какой-либо текст. Поле *field_name* содержит строку с именем поля в спецификаторе формата. Например, для спецификатора `'{0:d}'` именем поля будет `'0'`. Поле *format_spec* содержит спецификатор формата, следующий за двоеточием, например в предыдущем примере это поле будет содержать `'d'`. Может быть пустой строкой, если спецификатор не указан. Поле *conversion* содержит спецификатор преобразования (если присутствует). В предыдущем примере это поле будет содержать `None`, но, если бы использовался спецификатор `'{0!s:d}'`, поле *conversion* содержало бы `'s'`. Для последнего фрагмента строки формата все три поля *field_name*, *format_spec* и *conversion* будут содержать `None`.

f`.get_field(`*fieldname*`, `*args*`, `*kwargs*`)`

Извлекает из аргументов *args* и *kwargs* значение, ассоциированное с указанным именем поля *fieldname*. В аргументе *fieldname* передается строка, такая как `"0"` или `"name"`, которая возвращается методом `parse()`, представленным

выше, в поле *field_name*. Возвращает кортеж (*value*, *key*), где *value* – значение поля *fieldname*, а *key* – ключ, использовавшийся для поиска значения в аргументах *args* или *kwargs*. Если *key* – целое число, оно интерпретируется как индекс в кортеже *args*. Если строка, она интерпретируется как ключ словаря *kwargs*. Аргумент *fieldname* может включать дополнительные операции доступа к элементам последовательностей или к атрибутам, такие как '0.name' или '0[name]'. В этом случае метод выполнит обращение к требуемому элементу и вернет соответствующее значение. Однако значение *key* в возвращаемом кортеже при этом будет иметь значение '0'.

f.get_value(*key*, *args*, *kwargs*)

Извлекает объект из аргумента *args* или *kwargs*, в зависимости от типа значения *key*. Если *key* – целое число, объект извлекается из аргумента *args*. Если строка – из аргумента *kwargs*.

f.check_unused_args(*used_args*, *args*, *kwargs*)

Проверяет аргументы, не использованные в вызове format(). В аргументе *used_args* передается множество всех использованных ключей (смотрите описание get_field()), которые были найдены в строке форматирования. В аргументах *args* и *kwargs* передаются позиционные и именованные аргументы, полученные функцией format(). По умолчанию возбуждает исключение TypeError, если будут обнаружены неиспользованные аргументы.

f.format_value(*value*, *format_spec*)

Форматирует единственное значение *value* в соответствии с указанным спецификатором формата *format_spec*. По умолчанию просто вызывает встроенную функцию format(*value*, *format_spec*).

f.convert_field(*value*, *conversion*)

Преобразует значение *value*, полученное в результате вызова метода get_field(), в соответствии с указанным кодом преобразования *conversion*. Если в аргументе *conversion* передается None, возвращает оригинальное значение *value*. Если указан код преобразования 's' или 'r', значение *value* преобразуется в строку вызовом функции str() или repr() соответственно.

Если потребуется реализовать нестандартное форматирование строк, можно создать объект класса Formatter и просто использовать методы по умолчанию для реализации собственного типа форматирования. Можно также определить новый класс, производный от класса Formatter, и переопределить методы, описанные выше.

Дополнительные подробности о синтаксисе спецификаторов формата и о форматировании строк приводятся в главах 3 и 4.

Строки типа Template

В модуле string объявляется новый строковый тип Template, который упрощает некоторые операции подстановки. Пример использования этих строк приводится в главе 9.

Ниже показано, как создать новый объект строки шаблона:

```
Template(s)
```

Здесь *s* – это строка, а `Template` – имя класса.

Объект *t* класса `Template` поддерживает следующие методы:

```
t.substitute(m [, **kwargs])
```

Этот метод принимает объект отображения *m* (например, словарь) или список именованных аргументов и выполняет подстановку значений именованных аргументов в строку *t*. В процессе подстановки пары символов `'$$'` замещаются единственным символом `'$'`, а фрагменты `'$key'` или `'${key}'` – значениями `m['key']` или `kwargs['key']`, если методу были переданы именованные аргументы. Ключи *key* в строке должны быть допустимыми идентификаторами Python. Если в окончательной строке остались неопознанные ключи `'$key'`, возбуждается исключение `KeyError`.

```
t.safe_substitute(m [, **kwargs])
```

То же, что и `substitute()`, за исключением того, что этот метод не возбуждает исключение и не сообщает об ошибках. Неопознанные ключи *$key* остаются в строке в первоначальном виде.

```
t.template
```

Содержит оригинальную строку, переданную при вызове `Template()`.

Изменить поведение класса `Template` можно, создав производный класс и переопределив атрибуты `delimiter` и `idpattern`. Например, ниже приводится фрагмент, в котором символ экранирования `$` заменен на `@`, а в именах ключей позволяет использовать только алфавитные символы:

```
class MyTemplate(string.Template):
    delimiter = '@'      # Экранирующий символ
    idpattern = '[A-Z]*' # Шаблон идентификаторов
```

Вспомогательные функции

В модуле `string` имеется также пара функций для выполнения операций над строками, которые не являются строковыми методами.

```
capwords(s)
```

Переводит в верхний регистр первые буквы каждого слова в строке *s*, замещает повторяющиеся пробельные символы единственным пробелом и удаляет начальные и конечные пробелы.

```
maketrans(from, to)
```

Создает таблицу преобразования, которая отображает каждый символ в строке *from* в символ, находящийся в той же позиции, в строке *to*. Строки *from* и *to* должны иметь одинаковую длину. Эта функция используется для создания аргументов, пригодных для использования строковым методом `translate()`.

Модуль struct

Модуль `struct` используется для преобразования данных на языке Python в структуры двоичных данных (представленные как строки байтов) и наоборот. Такие структуры данных часто используются для организации взаимодействий с функциями, написанными на языке **C, при работе с двоичными** файлами, сетевыми протоколами или для обмена двоичными данными через последовательные порты.

Функции упаковывания и распаковывания

Ниже перечислены функции уровня модуля, используемые для упаковывания и распаковывания данных в строках байтов. Если в программе необходимо многократно выполнять эти операции, для этих целей можно использовать объект класса `Struct`, описываемый в следующем разделе.

`pack(`*`fmt`*`, `*`v1`*`, `*`v2`*`, ...)`

Упаковывает значения *v1*, *v2* и так далее, в строку байтов, в соответствии со строкой формата *fmt*.

`pack_into(`*`fmt`*`, `*`buffer`*`, `*`offset`*`, `*`v1`*`, `*`v2`*` ...)`

Упаковывает значения *v1*, *v2* и так далее, в объект буфера, доступный для записи, начиная с позиции *offset*. Этот метод может действовать только с объектами, поддерживающими интерфейс буферов. В качестве примеров таких объектов можно назвать `array.array` и `bytearray`.

`unpack(`*`fmt`*`, `*`string`*`)`

Распаковывает содержимое строки байтов *string* в соответствии со строкой формата *fmt*. Возвращает кортеж распакованных значений. Длина строки *string* должна точно соответствовать размеру формата, который определяется функцией `calcsize()`.

`unpack_from(`*`fmt`*`, `*`buffer`*`, `*`offset`*`)`

Распаковывает содержимое объекта *buffer* в соответствии со строкой формата *fmt*, начиная с позиции *offset*. Возвращает кортеж распакованных значений.

`calcsize(`*`fmt`*`)`

Вычисляет размер в байтах структуры, соответствующей строке формата *fmt*.

Объекты типа Struct

Модуль `struct` объявляет класс `Struct`, который предоставляет альтернативный интерфейс для операций упаковывания и распаковывания. Этот класс обеспечивает более высокую эффективность, так как строка формата интерпретируется всего один раз.

`Struct(`*`fmt`*`)`

Создает экземпляр класса `Struct`, представляющий данные, упакованные в соответствии с указанным кодом формата. Экземпляр *s* класса `Struct` об-

ладает следующими методами и действует так же, как и функции, описанные в предыдущем разделе:

Метод	Описание
s.pack(*v1*, *v2*, ...)	Упаковывает значения в строку байтов
s.pack_into(*buffer*, *offset*, *v1*, *v2*, ...)	Упаковывает значения в объект буфера
s.unpack(*bytes*)	Распаковывает значения из строки байтов
s.unpack_from(*buffer*, *offset*)	Распаковывает значения из объекта буфера
s.format	Используемый формат
s.size	Размер формата в байтах

Коды форматов

Строки форматов, используемые в модуле `struct`, являются последовательностями символов, которые имеют следующие значения:

Формат	Тип в языке C	Тип в языке Python
'x'	пустой байт	Не имеет значения
'c'	char	Строка длиной 1
'b'	signed char	Целое
'B'	unsigned char	Целое
'?'	_Bool (C99)	Логическое значение
'h'	short	Целое
'H'	unsigned short	Целое
'i'	int	Целое
'I'	unsigned int	Целое
'l'	long	Целое
'L'	unsigned long	Целое
'q'	long long	Длинное целое
'Q'	unsigned long long	Длинное целое
'f'	float	С плавающей точкой
'd'	double	С плавающей точкой
's'	char[]	Строка
'p'	char[]	Строка с длиной в первом байте
'P'	void *	Целое

Каждому символу формата может предшествовать целое число, определяющее количество повторений (например, '4i' – это то же, что и 'iiii'). Для

формата 's' число обозначает максимальную длину строки, то есть формат '10s' обозначает строку длиной до 10 байтов. Формат '0s' обозначает строку нулевой длины. Формат 'p' используется для представления строк, длина которых определяется значением в первом байте, за которым следует сама строка. Этот формат может пригодиться для организации взаимодействий с программным кодом, написанным на языке Pascal, что иногда бывает необходимо в системе Macintosh. Обратите внимание, что длина таких строк не может превышать 255 символов.

Когда для распаковывания двоичных данных используется формат 'I' или 'L', возвращается длинное целое число. Кроме того, формат 'P' может возвращать целое или длинное целое число, в зависимости от размера машинного слова.

Первый символ в строке формата может также определять порядок следования байтов и выравнивание в упакованных данных, как показано ниже:

Формат	Порядок следования байтов	Размер и выравнивание
'@'	Определяется аппаратной архитектурой	Определяется аппаратной архитектурой
'='	Определяется аппаратной архитектурой	Стандартные
'<'	Обратный порядок	Стандартные
'>'	Прямой порядок	Стандартные
'!'	Сетевой порядок (прямой)	Стандартные

Порядок следования байтов, определяемый аппаратной архитектурой, может быть прямым или обратным. Размеры и выравнивание, определяемые аппаратной архитектурой, соответствуют значениям, используемым компилятором языка C, и зависят от реализации. Под стандартным выравниванием подразумевается отсутствие необходимости выравнивания для любого типа. Под стандартным размером подразумевается, что значение типа `short` имеет размер 2 байта, значение типа `int` – 4 байта, значение типа `long` – 4 байта, значение типа `float` – 32 бита и значение типа `double` – 64 бита. Для формата 'P' может использоваться только порядок следования байтов, определяемый аппаратной архитектурой.

Примечания

Иногда бывает необходимо выравнивать конец структуры в соответствии с требованиями, предъявляемыми определенным типом. Для этого в конец строки формата следует добавить код формата требуемого типа с числом повторений, равным нулю. Например, формат 'llh0l' указывает, что структура должна заканчиваться на границе 4-байтового слова (предполагается, что значения типа `long` выравниваются по границам 4-байтовых слов). В данном случае после значения типа `short`, определяемого кодом 'h', будут добавлены два пустых байта. Этот прием действует, только когда указывается, что размер и выравнивание определяются аппаратной архитекту-

рой, – при стандартном подходе правила определения размера и выравнивания не действуют.

Форматы 'q' и 'Q' могут использоваться только в режиме «определяется аппаратной архитектурой», если компилятор языка C, использованный для сборки интерпретатора Python, поддерживает тип `long long`.

См. также

Описание модуля array (стр. 328) и модуля ctypes (стр. 759).

Модуль unicodedata

Модуль `unicodedata` обеспечивает доступ к базе данных символов Юникода, которая содержит информацию о свойствах всех символов Юникода.

`bidirectional(`*`unichr`*`)`

Возвращает признак направления письма для символа *unichr* в виде строки или пустую строку, если такое значение не определено. Возвращаемая строка может иметь одно из следующих значений:

Значение	Описание
L	Направление письма слева направо
LRE	Служебный символ, определяющий направление письма слева направо внутри текста, который записывается справа налево
LRO	Служебный символ, определяющий направление письма слева направо, независимо от направления письма в тексте
R	Направление письма справа налево
AL	Арабский символ, направление письма справа налево
RLE	Служебный символ, определяющий направление письма справа налево внутри текста, который записывается слева направо
RLO	Служебный символ, определяющий направление письма справа налево, независимо от направления письма в тексте
PDF	Служебный символ, отменяет предыдущее направление письма
EN	Европейское число
ES	Разделитель европейских чисел
ET	Конец европейского числа
AN	Арабское число
CS	Универсальный разделитель чисел
NSM	Знак, не занимающий дополнительного пространства
BN	Нейтральная граница
B	Разделитель абзацев

(продолжение)

Значение	Описание
S	Разделитель сегментов
WS	Пробельный символ
ON	Прочие нейтральные символы

`category(`*`unichr`*`)`

Возвращает строку, описывающую основную категорию, к которой относится символ *unichr*. Возвращаемая строка может иметь одно из следующих значений:

Значение	Описание
Lu	Буква, заглавная
Ll	Буква, строчная
Lt	Буква, с которой начинается слово, когда только первая буква слова записывается как заглавная
Mn	Знак, не занимающий дополнительного пространства
Mc	Знак, занимающий дополнительное пространство
Me	Объемлющий знак
Nd	Число, десятичная цифра
Nl	Число, буква
No	Число, прочие символы
Zs	Разделитель, пробельный символ
Zl	Разделитель строк
Zp	Разделитель абзацев
Cc	Управляющий символ
Cf	Неотображаемый символ, признак форматирования
Cs	Прочие, суррогатная пара
Co	Прочие, предназначен для закрытого использования
Cn	Прочие, символ не присвоен
Lm	Буква, модификатор
Lo	Буква, прочие
Pc	Знак пунктуации, соединяющий слова
Pd	Знак пунктуации, дефис
Ps	Знак пунктуации, открывающая скобка
Pe	Знак пунктуации, закрывающая скобка
Pi	Знак пунктуации, открывающие кавычки

Значение	Описание
Pf	Знак пунктуации, закрывающие кавычки
Po	Знак пунктуации, прочие
Sm	Символ, математический
Sc	Символ, знак денежной единицы
Sk	Символ, модификатор
So	Символ, прочие

`combining(`*`unichr`*`)`

Возвращает целое число, описывающее класс объединения для символа *unichr* или 0, если класс объединения не определен. Возвращается одно из следующих значений:

Значение	Описание
0	Пробел, разделитель, признак переупорядочения, дополнительный знак тибетского алфавита
1	Комбинационный знак; описывает или вписывается внутрь основного символа
7	Знак нюкта (Nukta, индийский алфавит)
8	Знак транскрипции для алфавита Хирагана/Катакана
9	Вирама
10-199	Классы фиксированных позиций
200	Присоединяется снизу слева
202	Присоединяется снизу
204	Присоединяется снизу справа
208	Присоединяется слева
210	Присоединяется справа
212	Присоединяется сверху слева
214	Присоединяется сверху
216	Присоединяется сверху справа
218	Снизу слева
220	Снизу
222	Снизу справа
224	Слева
226	Справа
228	Сверху слева
230	Сверху

(продолжение)

Значение	Описание
232	Сверху справа
233	Двойной снизу
234	Двойной сверху
240	Снизу (подстрочный знак)

`decimal(`*`unichr`* `[,` *`default`*`])`

Возвращает десятичное целое значение цифрового символа *unichr*. Если *unichr* не является десятичной цифрой, возвращается значение аргумента *default* или возбуждается исключение `ValueError`, если этот аргумент опущен.

`decomposition(`*`unichr`*`)`

Возвращает строку, содержащую разложение символа *unichr* или пустую строку, если разложение не определено. Обычно символы, содержащие знаки акцента, могут быть разложены в многосимвольные последовательности. Например, вызов `decomposition(u"\u00fc")` (символ "ь") вернет строку "0075 0308", соответствующую букве *u* и знаку акцента (*¨*). Строка, возвращаемая этой функцией, может также включать следующие строки:

Значение	Описание
<font>	Вариант шрифта (например, готический)
<noBreak>	Неразрывный пробел или дефис
<initial>	Форма начального представления (Арабский алфавит)
<medial>	Форма промежуточного представления (Арабский алфавит)
<final>	Форма окончательного представления (Арабский алфавит)
<isolated>	Форма изолированного представления (Арабский алфавит)
<circle>	Форма символов, заключенных в кружок
<super>	Надстрочная форма
<sub>	Подстрочная форма
<vertical>	Форма вертикального представления
<wide>	Широкий (или дзенкаку) символ совместимости (японский алфавит)
<narrow>	Узкий (или ханкаку) символ совместимости (японский алфавит)
<small>	Форма уменьшенного представления (для совместимости с китайскими символами)
<square>	Шрифт квадратной формы для китайского, японского и корейского языков
<fraction>	Простейшая дробь
<compat>	Прочие символы совместимости

`digit(`*`unichr`* `[, `*`default`*`])`

Возвращает целое значение цифрового символа *unichr*. Если *unichr* не является цифрой, возвращается значение аргумента *default* или возбуждается исключение `ValueError`, если этот аргумент опущен. Эта функция отличается от функции `decimal()` тем, что может принимать символы, которые могут представлять цифры, но не являются десятичными цифрами.

`east_asian_width(`*`unichr`*`)`

Возвращает строку, обозначающую ширину символа *unichr* для восточноазиатских языков.

`lookup(`*`name`*`)`

Отыскивает символ по его имени. Например, вызов `lookup('COPYRIGHT SIGN')` вернет соответствующий символ Юникода. Наиболее распространенные имена символов можно найти по адресу *http://www.unicode.org/charts*.

`mirrored(`*`unichr`*`)`

Возвращает `1`, если символ *unichr* является зеркально отражаемым при двунаправленном письме, в противном случае возвращает `0`. Зеркально отражаемым называется тот символ, чей внешний вид должен изменяться при отображении текста в обратном порядке. Например, символ `'('` является зеркально отражаемым, потому что при выводе текста справа налево его следует отразить в символ `')'`.

`name(`*`unichr`* `[, `*`default`*`])`

Возвращает имя символа Юникода *unichr*. Возбуждает исключение `ValueError`, если имя не определено, или возвращает значение аргумента *default*, если он не был опущен. Например, вызов `name(u'\xfc')` вернет `'LATIN SMALL LETTER U WITH DIAERESIS'`.

`normalize(`*`form`*`, `*`unistr`*`)`

Нормализует строку Юникода *unistr* в соответствии с нормальной формой *form*. В аргументе *form* можно передать строку `'NFC'`, `'NFKC'`, `'NFD'` или `'NFKD'`. Нормализация строки в определенной степени связана с композицией и декомпозицией определенных символов. Например, строка Юникода со словом «resumй» может быть представлена как `u'resum\u00e9'` или как `u'resume\u0301'`. В первом случае акцентированный символ *й* представлен одним символом, а во втором он представлен буквой *e*, за которой следует присоединяемый знак акцента (*´*). При использовании формы нормализации `'NFC'` строка *unistr* будет преобразована так, что будет содержать только скомпонованные символы (так, символ й будет представлен единственным символом). При использовании формы нормализации `'NFD'` строка *unistr* будет преобразована так, что будет содержать комбинационные знаки (например, символ й будет представлен буквой *e*, за которой следует знак акцента). При использовании форм `'NFKC'` и `'NFKD'` нормализация выполняется так же, как и при использовании форм `'NFC'` и `'NFD'`, за исключением того, что в этом случае дополнительно выполняется преобразование некоторых символов, которые могут быть представлены больше чем одним символом Юникода. Например, римским цифрам соответствуют собственные символы Юникода, но они также могут быть представлены символами латинско-

го алфавита I, V, M и так далее. При использовании форм 'NFKC' и 'NFKD' специальные символы римских цифр будут преобразованы в их эквиваленты с использованием символов латинского алфавита.

numeric(*unichr* [, *default*])

Возвращает числовое значение символа Юникода в виде числа с плавающей точкой. Если числовое значение не определено, возвращает значение аргумента *default* или возбуждает исключение ValueError. Например, символ U+2155 (символ, изображающий дробь "1/5") имеет числовое значение 0.2.

unidata_version

Строка, содержащая номер версии базы данных символов Юникода (например, '5.1.0').

Примечание

Дополнительные сведения о базе данных символов Юникода можно найти по адресу: *http://www.unicode.org*.

17

Доступ к базам данных

В этой главе описываются программные интерфейсы, которые используются программами на языке Python с реляционными базами данных и с базами данных, подобных хеш-таблицам. В отличие от других глав, где описываются модули из стандартной библиотеки, в этой главе будут затронуты сторонние расширения. Например, если вам потребуется организовать доступ к базе данных MySQL или Oracle, для начала вам придется загрузить модуль стороннего расширения. Но этот модуль, в свою очередь, будет следовать основным соглашениям, которые описываются здесь.

Прикладной интерфейс доступа к реляционным базам данных

Для обеспечения доступа к реляционным базам данных сообществом разработчиков **Python был выработан стандарт, известный как «Python Database API Specification V2.0»**, или PEP 249 (формальное описание можно найти по адресу *http://www.python.org/dev/peps/pep-249/*). Модули для доступа к определенным базам данных (таким как MySQL, Oracle и другие) следуют этому стандарту, но могут добавлять дополнительные возможности. В этом разделе рассматриваются самые основные элементы интерфейса, необходимые большинству приложений.

На верхнем уровне прикладного интерфейса доступа к базам данных определяется множество функций и классов, обеспечивающих соединение с сервером баз данных, выполнение запросов SQL и получение результатов. Для этих целей используются два основных класса: `Connection`, который управляет соединением с базой данных, и `Cursor`, используемый для выполнения запросов.

Соединения

Каждый модуль доступа к базам данных предоставляет функцию `connect(`*`parameters`*`)`, позволяющую установить соединение с базой данных.

Точное количество аргументов зависит от базы данных, однако в их число обычно входят: имя источника данных, имя пользователя, пароль, имя хоста и имя базы данных. Как правило, все эти значения передаются в виде именованных аргументов `dsn`, `user`, `password`, `host` и `database` соответственно. Поэтому вызов `connect()` может выглядеть, как показано ниже:

```
connect(dsn="hostname:DBNAME",user="michael",password="peekaboo")
```

В случае успеха функция возвращает объект класса `Connection`. Экземпляр *`c`* класса `Connection` обладает следующими методами:

`c``.close()`

Закрывает соединение с сервером.

`c``.commit()`

Подтверждает все незавершенные транзакции в базе данных. Если база данных поддерживает механизм транзакций, любые изменения в базе данных вступят в силу только после вызова этого метода. Если база данных не поддерживает транзакции, этот метод ничего не делает.

`c``.rollback()`

Откатывает все изменения в базе данных до момента, когда были запущены какие-либо незавершенные транзакции. Иногда этот метод используется при работе с базами данных, не поддерживающими транзакции, чтобы отменить любые изменения, произведенные в них. Например, если в процессе изменения данных в базе возникло исключение, можно воспользоваться этим методом, чтобы отменить изменения, произведенные до появления исключения.

`c``.cursor()`

Создает новый курсор, экземпляр класса `Cursor`, использующий соединение. Курсор – это объект, который используется для выполнения запросов SQL и получения результатов. Об этом рассказывается в следующем разделе.

Курсоры

Чтобы выполнить какие-либо операции в базе данных, сначала необходимо создать объект соединения *`c`*, а затем вызовом метода *`c`*`.cursor()` создать объект класса `Cursor`. Экземпляр *`cur`* класса `Cursor` обладает множеством стандартных методов и атрибутов, которые используются для выполнения запросов:

`cur``.callproc(`*`procname`* `[,` *`parameters`*`])`

Вызывает хранимую процедуру с именем *`procname`*. В аргументе *`parameters`* передается последовательность значений, которые будут переданы хранимой процедуре в виде ее аргументов. Результатом этой функции является последовательность с тем же количеством элементов, что и в последовательности *`parameters`*. Эта последовательность является копией последовательности *`parameters`*, где значения любых выходных аргументов замещаются значениями, полученными в результате выполнения хранимой про-

цедуры. Кроме того, если процедура выводит некоторый набор данных, его можно получить с помощью методов `fetch*()`, описываемых ниже.

cur`.close()`

Закрывает курсор, предотвращая возможность выполнения каких-либо дальнейших операций с его помощью.

cur`.execute(`*query* `[, ` *parameters*`])`

Выполняет запрос или команду *query* в базе данных. В аргументе *query* передается строка, содержащая команду (обычно на языке SQL), а в аргументе *parameters* — либо последовательность, либо отображение со значениями переменных, используемых в строке запроса *query* (об этом рассказывается в следующем разделе).

cur`.executemany(`*query* `[, ` *parametersequence*`])`

Многократно выполняет запрос или команду *query*. В аргументе *query* передается строка запроса, а в аргументе *parametersquence* — последовательность групп параметров. Каждый элемент этой последовательности должен быть объектом последовательности или отображения, который можно было бы передать методу `execute()`, описанному выше.

cur`.fetchone()`

Возвращает следующую запись из набора данных, полученного вызовом метода `execute()` или `executemany()`. Как правило, результатом является список или кортеж значений различных столбцов в наборе данных. После извлечения последней записи из набора возвращается `None`. В случае отсутствия набора данных или в случае, когда предыдущая операция не вернула ничего, возбуждается исключение.

cur`.fetchmany([`*size*`])`

Возвращает последовательность записей из полученного набора данных (то есть список кортежей). В аргументе *size* указывается количество записей, которые требуется вернуть. При вызове без аргумента в качестве значения по умолчанию используется значение атрибута *cur*`.arraysize`. Фактическое число возвращаемых записей может оказаться меньше запрошенного. Если ранее были выбраны все записи, возвращается пустая последовательность.

cur`.fetchall()`

Возвращает последовательность всех записей (то есть список кортежей), оставшихся в полученном наборе данных.

cur`.nextset()`

Пропускает все оставшиеся записи в текущем наборе данных и переходит к следующему набору (если имеется). Если следующего набора данных не существует, возвращает `None`; в противном случае возвращает `True`, а последующие операции `fetch*()` будут возвращать данные из нового набора.

cur`.setinputsize(`*sizes*`)`

Сообщает курсору информацию о параметрах, которые будут передаваться в последующих вызовах методов `execute*()`. В аргументе *sizes* передается

последовательность объектов типов (описывается чуть ниже) или целых чисел, которые обозначают максимальную ожидаемую длину строки для каждого параметра. Эта информация используется объектом курсора для выделения буферов в памяти, которые будут использоваться при создании запросов и команд, посылаемых базе данных. Это позволяет повысить скорость выполнения последующих операций `execute*()`.

cur.`setoutputsize(`*size* `[, ` *column*`])`

Устанавливает размер буфера для определенного столбца в возвращаемом наборе данных. В аргументе *column* передается целочисленный индекс поля в записи, а в аргументе *size* – количество байтов. Обычно этот метод используется, чтобы установить ограничения для столбцов, которые могут содержать большие объемы информации, таких как строки, объекты BLOB и объекты LONG, перед тем, как вызывать методы `execute*()`. Если аргумент *column* опущен, ограничение *size* устанавливается для всех столбцов в возвращаемом наборе данных.

Курсоры обладают рядом атрибутов с информацией о текущем наборе данных и о самом курсоре.

cur.`arraysize`

Целое число, которое используется методом `fetchmany()` как значение по умолчанию. Это значение может отличаться в разных модулях доступа к базам данных и изначально может устанавливаться равным «оптимальному», с точки зрения модуля, значению.

cur.`description`

Последовательность кортежей с информацией о каждом столбце в текущем наборе данных. Каждый кортеж имеет вид (*name*, *type_code*, *display_size*, *internal_size*, *precision*, *scale*, *null_ok*). Первое поле всегда содержит имя столбца. Значение в поле *type_code* может использоваться в операциях сравнивания с типами объектов, о которых рассказывается в разделе «Типы объектов». Другие поля в кортеже могут иметь значение `None`, если они не имеют смысла для столбца.

cur.`rowcount`

Количество записей в наборе данных, полученном в результате последнего вызова одного из методов `execute*()`. Значение `-1` означает, что либо набор данных в результате отсутствует, либо количество записей не может быть определено.

Хотя это и не оговаривается спецификацией, тем не менее реализация классов `Cursor` в большинстве модулей доступа к базам данных обеспечивают поддержку протокола итераций. То есть, чтобы обойти все записи в наборе данных, полученном в результате последнего вызова одного из методов `execute*()`, можно использовать инструкцию, такую как `for` *row* `in` *cur*:.

Ниже приводится простой пример, демонстрирующий, как некоторые из этих операций могут использоваться при работе с модулем `sqlite3` доступа к базе данных, который входит в состав стандартной библиотеки:

```
import sqlite3
conn = sqlite3.connect("dbfile")
cur = conn.cursor()

# Пример простого запроса
cur.execute("select name, shares, price from portfolio where account=12345")

# Обход результатов в цикле
while True:
    row = cur.fetchone()
    if not row: break
    # Обработать запись
    name, shares, price = row
    ...

# Альтернативный подход (с использованием итераций)
cur.execute("select name, shares, price from portfolio where account=12345")
for name, shares, price in cur:
    # Обработать запись
    ...
```

Формирование запросов

Критически важным этапом при использовании прикладного интерфейса доступа к базам данных является формирование строки запроса SQL, которая передается методам `execute*()` объектов курсоров. Отчасти проблема заключается в необходимости вставить в строку запроса параметры, которые вводятся пользователем. Например, можно было бы написать такой программный код:

```
symbol = "AIG"
account = 12345

cur.execute("select shares from portfolio where name='%s' and account=%d" %
            (symbol, account))
```

Этот прием «действует», тем не менее никогда не следует вручную формировать запросы с помощью строковых операций, как это сделано в данном примере. В противном случае появляется уязвимость к атакам типа «инъекция SQL», которой может воспользоваться злоумышленник, чтобы выполнить произвольные инструкции на сервере баз данных. Например, в предыдущем примере злоумышленник мог бы передать в параметре `symbol` значение `"EVIL LAUGH'; drop table portfolio;--"` которое, очевидно, приводит совсем не к тем результатам, которые вы могли бы ожидать.

Все модули доступа к базам данных предоставляют собственные механизмы подстановки значений. Например, вместо того, чтобы формировать запрос целиком, как было показано выше, то же самое можно было бы сделать иначе:

```
symbol = "AIG"
account = 12345

cur.execute("select shares from portfolio where name=? and account=?",
            (symbol, account))
```

Здесь символы подстановки '?' благополучно замещаются значениями из кортежа (symbol, account).

К сожалению, не существует единого соглашения о правилах оформления символов подстановки между различными модулями доступа к базам данных. При этом в каждом модуле может быть задана переменная paramstyle, которая определяет формат параметров в запросах, замещаемых фактическими значениями. Ниже перечислены возможные значения этой переменной:

Формат параметра	Описание
'qmark'	Параметры обозначаются знаком вопроса, где каждый символ ? в запросе замещается очередным элементом последовательности. Например: *cur*.execute("... where name=? and account=?", (symbol, account)). Значения параметров передаются в виде кортежа.
'numeric'	Параметры обозначаются числами, где параметр :*n* замещается значением с индексом *n*. Например: *cur*.execute("... where name=:0 and account=:1",(symbol, account)).
'named'	Именованные параметры, где параметр :*name* замещается именованным значением. В этом случае значения параметров должны передаваться в виде отображения. Например: *cur*.execute("... where name=:symbol and account=:account", {'symbol':symbol, 'account': account}).
'format'	Параметры обозначаются в стиле функции printf, в виде спецификаторов формата, таких как %s, %d и так далее. Например: *cur*.execute("... where name=%s and account=%d", (symbol, account)).
'pyformat'	Расширенный набор кодов формата в стиле языка Python, таких как %(name)s. Напоминает формат 'named'. Значения параметров должны передаваться в виде отображения, а не в виде кортежа.

Типы объектов

При работе с базами данных встроенные типы, такие как целые числа и строки, обычно отображаются в эквивалентные типы, используемые базой данных. Однако все усложняется, когда приходится работать с такими данными, как даты, двоичные данные, и другими специальными типами. Чтобы упростить отображение данных этих типов, модули доступа к базам данных реализуют набор функций-конструкторов, позволяющих создавать объекты различных типов.

Date(*year*, *month*, *day*)

Создает объект, представляющий дату.

Time(*hour*, *minute*, *second*)

Создает объект, представляющий время.

`Timestamp(`*`year`*`, `*`month`*`, `*`day`*`, `*`hour`*`, `*`minute`*`, `*`second`*`)`

Создает объект, представляющий отметку времени.

`DateFromTicks(`*`ticks`*`)`

Создает объект, представляющий дату, из значения системного времени. В аргументе *`ticks`* передается количество секунд, возвращаемое такими функциями, как `time.time()`.

`TimeFromTicks(`*`ticks`*`)`

Создает объект, представляющий время, из значения системного времени.

`TimestampFromTicks(`*`ticks`*`)`

Создает объект, представляющий отметку времени, из значения системного времени.

`Binary(`*`s`*`)`

Создает объект, представляющий двоичные данные, из строки байтов *`s`*.

В дополнение к этим конструкторам в модулях могут объявляться следующие типы объектов. Эти типы объектов предназначены для проверки типа, указанного в поле *`type_code`* атрибута *`cur`*`.description`, который содержит информацию о содержимом текущего набора данных.

Тип объекта	Описание
`STRING`	Символьные или текстовые данные
`BINARY`	Двоичные данные, такие как BLOB
`NUMBER`	Числовые данные
`DATETIME`	Дата и время
`ROWID`	Идентификатор записи

Обработка ошибок

Модули доступа к базам данных объявляют исключение `Error` верхнего уровня, которое служит базовым классом для всех остальных исключений. Ниже приводится перечень более специализированных исключений, имеющих отношение к ошибкам, возникающим при работе с базами данных:

Исключение	Описание
`InterfaceError`	Ошибки, связанные с неправильным использованием интерфейса к базе данных, но не с самой базой данных.
`DatabaseError`	Ошибки, имеющие отношение к самой базе данных.
`DataError`	Ошибки, связанные с обработкой данных. Например, недопустимое преобразование типов, деление на ноль и так далее.
`OperationalError`	Ошибки, связанные с работой самой базы данных. Например, потеря соединения.

(продолжение)

Исключение	Описание
`IntegrityError`	Ошибки, связанные с нарушением целостности базы данных.
`InternalError`	Внутренняя ошибка базы данных. Например, обращение к устаревшему курсору.
`ProgrammingError`	Ошибки в запросах SQL.
`NotSupportedError`	Ошибки обращения к методам программного интерфейса, которые не поддерживаются базой данных.

Модули могут также объявлять исключение `Warning`, которое используется модулями доступа к базам данных, чтобы предупредить о таких проблемах, как усечение данных при записи в базу данных.

Многопоточность

При разработке многопоточных приложений следует помнить, что модули доступа к базам данных могут поддерживать, а могут не поддерживать возможность работы в многопоточном режиме. Каждый модуль объявляет следующие переменные, позволяющие получить дополнительную информацию об этом.

`threadsafety`

Целое число, описывающее степень поддержки модулем работы в многопоточном режиме. Ниже перечислены возможные значения:

0 Работа в многопоточном режиме не поддерживается. Различные потоки не могут совместно использовать какие-либо элементы модуля.

1 Модуль поддерживает работу в многопоточном режиме, но соединения не могут совместно использоваться в разных потоках.

2 Модуль и соединения поддерживают работу в многопоточном режиме, но курсоры не могут совместно использоваться в разных потоках.

3 Модуль, соединения и курсоры поддерживают работу в многопоточном режиме.

Отображение результатов в словари

Часто возникает необходимость отобразить результаты, полученные из базы данных в виде кортежей или списков, в словарь с именованными полями. Например, если набор данных, полученный в результате запроса, содержит большое число столбцов, с ним было бы проще работать, используя описательные имена полей вместо жестко заданных числовых индексов в кортеже.

Решить эту проблему можно множеством способов, но один из самых элегантных заключается в использовании функций-генераторов. Например:

```
def generate_dicts(cur):
    import itertools
    fieldnames = [d[0].lower() for d in cur.description ]
    while True:
        rows = cur.fetchmany()
        if not row: return
        for row in rows:
            yield dict(itertools.izip(fieldnames,row))

# Пример использования
cur.execute("select name, shares, price from portfolio")
for r in generate_dicts(cur):
    print r['name'],r['shares'],r['price']
```

Имейте в виду, что способ именования столбцов в разных базах данных может отличаться, особенно это касается чувствительности к регистру символов. Поэтому вам придется быть достаточно внимательными при попытке применить этот прием в программном коде, который должен обеспечивать возможность работы с различными модулями доступа к базам данных.

Расширение интерфейса доступа к базам данных

В заключение следует отметить, что к конкретным модулям баз данных может быть добавлено множество расширений и дополнительных возможностей, например поддержка двухфазного подтверждения транзакций или расширенная обработка ошибок. Дополнительную информацию о рекомендуемых интерфейсах подобных улучшений можно найти в документе PEP-249. Кроме того, сторонние модули могут существенно упростить работу с интерфейсами доступа к реляционным базам данных.

Модуль sqlite3

Модуль `sqlite3` реализует интерфейс к библиотеке базы данных SQLite (*http://www.sqlite.org*). SQLite – это библиотека на языке C, которая реализует функциональность реляционной базы данных, размещаемой в едином файле или в области памяти. Несмотря на свою простоту, эта библиотека может оказаться весьма привлекательной по различным причинам. Во-первых, она не требует выделения специального сервера баз данных и не требует сложной настройки – ее можно начинать использовать, просто установив соединение с файлом базы данных (если этот файл отсутствует, будет создан новый). Кроме всего прочего, эта база данных поддерживает транзакции, что повышает ее надежность (даже в случае системных сбоев), а также имеет механизм блокировок, что позволяет одновременно работать с одним и тем же файлом базы данных из нескольких процессов.

Программный интерфейс к библиотеке следует соглашениям, описанным в предыдущем разделе, посвященном интерфейсу доступа к базам данных, поэтому здесь мы не будем повторять описание многих особенностей. Вместо этого в данном разделе мы сосредоточимся на технических подробностях использования модуля, а также на особенностях, характерных для модуля `sqlite3`.

Функции уровня модуля

В модуле sqlite3 определены следующие функции:

connect(*database* [, *timeout* [, *isolation_level* [, *detect_types*]]])

Создает соединение с базой данных SQLite. В аргументе *database* передается строка, определяющая имя файла базы данных. В нем также можно передать строку ":memory:"; в этом случае база данных будет создана в памяти (имейте в виду, что база данных этого типа существует, только пока выполняется программа на языке Python, и исчезает сразу после завершения программы). Аргумент *timeout* определяет интервал времени ожидания освобождения внутренней блокировки на доступ для чтения-записи, пока другие соединения выполняют изменения данных в базе. По умолчанию предельное время ожидания составляет 5 секунд. Когда используются такие операторы SQL, как INSERT или UPDATE, автоматически запускается новая транзакция, если она не была запущена ранее. В аргументе *isolation_level* передается строка с необязательным дополнительным модификатором инструкции SQL BEGIN, которая используется для запуска транзакции. Возможные значения: "" (по умолчанию), "DEFERRED", "EXCLUSIVE" или "IMMEDIATE". **Эти модификаторы имеют отношение к реализации механизма блокировок в базе данных и имеют следующий смысл:**

Уровень изоляции	Описание
"" (пустая строка)	Используется значение по умолчанию – DEFFERED.
"DEFFERED"	Выполняется запуск новой транзакции, но блокировка устанавливается только в момент выполнения первой операции.
"EXCLUSIVE"	Запускает новую транзакцию и гарантирует, что никакое другое соединение с базой данных не сможет выполнять операции чтения или записи, пока не будет подтверждена эта транзакция.
"IMMEDIATE"	Запускает новую транзакцию и гарантирует, что никакое другое соединение с базой данных не сможет выполнять изменения, пока не будет подтверждена эта транзакция. Однако при этом остальные соединения смогут выполнять операции чтения из базы данных.

Аргумент *detect_types* позволяет организовать дополнительное определение типов данных (за счет дополнительного синтаксического анализа запросов SQL) в возвращаемых наборах данных. По умолчанию имеет значение 0 (дополнительное определение не производится). Может устанавливаться как битная маска, составленная из флагов PARSE_DECLTYPES и PARSE_COLNAMES с помощью битовой операции ИЛИ. Если установлен флаг PARSE_DECLTYPES, запросы проверяются на наличие в них имен типов, таких как "integer" или "number(8)", чтобы определить типы столбцов в возвращаемом наборе данных. Если установлен флаг PARSE_COLNAMES, появляется возможность встраивать в запросы специальные строки вида "*colname* [*typename*]" (вклю-

чая кавычки), где *colname* – имя столбца, а *typename* – имя типа, зарегистрированного с помощью функции register_converter(), описываемой ниже. При передаче механизму SQLite эти строки просто преобразуются в строки *colname*, а дополнительный спецификатор типа используется при преобразовании значений, полученных в результате запроса. Например, такой запрос, как 'select price as "price [decimal]" from portfolio', будет интерпретироваться как 'select price as price from portfolio', а результаты будут преобразованы в соответствии с правилом «decimal».

register_converter(*typename*, *func*)

Регистрирует новое имя типа для последующего использования в качестве значения аргумента *detect_types* функции connect(). В аргументе *typename* передается строка, содержащая имя типа в том виде, в каком оно будет использоваться в запросах, а в аргументе *func* – функция, принимающая единственную строку байтов и возвращающая тип данных в языке Python. Так, если выполнить вызов sqlite3.register_converter('decimal', decimal.Decimal), появится возможность запрашивать значения, которые будут преобразованы в объекты Decimal, например: 'select price as "price [decimal]" from stocks'.

register_adapter(*type*, *func*)

Регистрирует функцию преобразования типа *type* в языке Python, которая будет использоваться при попытке сохранить значение этого типа в базе данных. В аргументе *func* передается функция, принимающая экземпляр типа *type* и возвращающая значение типа int, float, строку байтов в кодировке UTF-8, строку Юникода или буфер. Например, если потребуется сохранить в базе данных объект типа Decimal, это можно реализовать вызовом sqlite3.register_adapter(decimal.Decimal,float).

complete_statement(*s*)

Возвращает True, если строка *s* содержит одну или более полных инструкций на языке SQL, разделенных точкой с запятой. Это может пригодиться при разработке интерактивных программ, которые получают запросы от пользователя.

enable_callback_tracebacks(*flag*)

Определяет необходимость обработки исключений, возникших в пользовательских функциях обратного вызова, таких как функции, зарегистрированные с помощью функций register_converter() и register_adapter(). По умолчанию исключения игнорируются. Если в аргументе *flag* передать значение True, сообщения с трассировочной информацией будут выводиться в поток sys.stderr.

Объекты класса Connection

Объект *c* класса Connection возвращается функцией connect() и поддерживает стандартные операции, описанные в разделе, посвященном описанию интерфейса доступа к базам данных. В дополнение к ним предоставляются следующие методы, характерные для модуля sqlite3.

c.create_function(*name*, *num_params*, *func*)

Регистрирует пользовательскую функцию для использования в запросах SQL. В аргументе *name* передается строка с именем функции, в аргументе *num_params* – целое число, определяющее количество параметров, и в аргументе *func* – функция на языке Python, содержащая реализацию. Ниже приводится простой пример:

```
def toupper(s):
    return s.upper()
c.create_function("toupper", 1, toupper)

# Пример использования в запросе
c.execute("select toupper(name), foo, bar from sometable")
```

Даже при том, что функция записывается на языке Python, ее аргументами могут быть только значения типов `int`, `float`, `str`, `unicode`, `buffer` или `None`.

c.create_aggregate(*name*, *num_params*, *aggregate_class*)

Регистрирует пользовательскую агрегатную функцию для использования в запросах SQL. В аргументе *name* передается строка с именем функции, а в аргументе *num_params* – целое число, определяющее количество входных параметров. В аргументе *aggregate_class* указывается класс, реализующий агрегатную операцию. Этот класс должен поддерживать инициализацию без параметров и реализовать метод `step(`*params*`)`, принимающий то же число параметров, которое указано в аргументе *num_params*, и метод `finalize()`, возвращающий окончательный результат. Ниже приводится простой пример:

```
class Averager(object):
    def __init__(self):
        self.total = 0.0
        self.count = 0
    def step(self,value):
        self.total += value
        self.count += 1
    def finalize(self):
        return self.total / self.count

c.create_aggregate("myavg", 1, Averager)

# Пример использования в запросе
c.execute("select myavg(num) from sometable")
```

Агрегатная операция выполняется как последовательность вызовов метода `step()` с входными значениями и последующим вызовом метода `finalize()` для получения окончательного значения.

c.create_collation(*name*, *func*)

Регистрирует пользовательскую функцию сравнивания для использования в запросах SQL. В аргументе *name* передается строка с именем функции сравнивания, а в аргументе *func* – функция, принимающая два аргумента и возвращающая значение -1, 0 или 1, в зависимости от того, является ли

первый аргумент меньшим, равным или большим второго. Пользовательская функция может использоваться в выражениях на языке SQL, таких как "select * from table order by colname collate name".

c.execute(*sql* [, *params*])

Обеспечивает упрощенный способ создания объекта курсора с помощью метода *c*.cursor() с последующим вызовом метода execute() курсора. В аргументе *sql* передается запрос SQL, а в аргументе *params* – параметры запроса.

c.executemany(*sql* [, *params*])

Обеспечивает упрощенный способ создания объекта курсора с помощью метода *c*.cursor() с последующим вызовом метода executemany() курсора. В аргументе *sql* передается запрос SQL, а в аргументе *params* – параметры запроса.

c.executescript(*sql*)

Обеспечивает упрощенный способ создания объекта курсора с помощью метода *c*.cursor() с последующим вызовом метода executescript() курсора. В аргументе *sql* передается запрос SQL.

c.interrupt()

Прерывает выполнение запросов, выполняющихся в данном соединении. Этот метод предназначен для вызова из другого потока выполнения.

c.iterdump()

Возвращает итератор, который выводит все содержимое базы данных в виде серии инструкций SQL, которые в свою очередь могут использоваться для воссоздания базы данных. Этот метод может пригодиться для экспортирования базы данных в другое место или при необходимости сохранить в файле базу данных, расположенную в памяти, для последующего восстановления.

c.set_authorizer(*auth_callback*)

Регистрирует функцию авторизации, которая вызывается при каждой попытке обратиться к столбцу данных в базе. Функция должна принимать пять аргументов, в виде: auth_callback(*code*, *arg1*, *arg2*, *dbname*, *innername*). Значение, возвращаемое этой функцией, может быть одним из следующих: SQLITE_OK – если доступ разрешен, SQLITE_DENY – если попытка доступа должна завершаться ошибкой, и SQLITE_IGNORE – если вместо фактического значения столбца должно возвращаться значение Null. В первом аргументе *code* функции будет передаваться целочисленный код операции. В аргументах *arg1* и *arg2* – параметры операции, значения которых зависят от кода операции. В аргументе *dbname* передается строка с именем базы данных (обычно "main"), в аргументе *innername* – имя внутреннего представления или триггера, осуществляющего попытку доступа, или None, если отсутствует активное представление или триггер. В следующей таблице приводится список значений кодов операций и интерпретация параметров *arg1* и *arg2*:

Код операции	arg1	arg2
SQLITE_CREATE_INDEX	Имя индекса	Имя таблицы
SQLITE_CREATE_TABLE	Имя таблицы	None
SQLITE_CREATE_TEMP_INDEX	Имя индекса	Имя таблицы
SQLITE_CREATE_TEMP_TABLE	Имя таблицы	None
SQLITE_CREATE_TEMP_TRIGGER	Имя триггера	Имя таблицы
SQLITE_CREATE_TEMP_VIEW	Имя представления	None
SQLITE_CREATE_TRIGGER	Имя триггера	Имя таблицы
SQLITE_CREATE_VIEW	Имя представления	None
SQLITE_DELETE	Имя таблицы	None
SQLITE_DROP_INDEX	Имя индекса	Имя таблицы
SQLITE_DROP_TABLE	Имя таблицы	None
SQLITE_DROP_TEMP_INDEX	Имя индекса	Имя таблицы
SQLITE_DROP_TEMP_TABLE	Имя таблицы	None
SQLITE_DROP_TEMP_TRIGGER	Имя триггера	Имя таблицы
SQLITE_DROP_TEMP_VIEW	Имя представления	None
SQLITE_DROP_TRIGGER	Имя триггера	Имя таблицы
SQLITE_DROP_VIEW	Имя представления	None
SQLITE_INSERT	Имя таблицы	None
SQLITE_PRAGMA	Имя директивы	None
SQLITE_READ	Имя таблицы	Имя столбца
SQLITE_SELECT	None	None
SQLITE_TRANSACTION	None	None
SQLITE_UPDATE	Имя таблицы	Имя столбца
SQLITE_ATTACH	Имя файла	None
SQLITE_DETACH	Имя базы данных	None
SQLITE_ALTER_TABLE	Имя базы данных	Имя таблицы
SQLITE_REINDEX	Имя индекса	None
SQLITE_ANALYZE	Имя таблицы	None
SQLITE_CREATE_VTABLE	Имя таблицы	Имя модуля
SQLITE_DROP_VTABLE	Имя таблицы	Имя модуля
SQLITE_FUNCTION	Имя функции	None

c.set_progress_handler(*handler*, *n*)

Регистрирует функцию обратного вызова, которая должна вызываться при выполнении каждой *n*-й инструкции виртуальной машины SQLite. В аргументе *handler* передается функция, не имеющая аргументов.

Ниже приводятся дополнительные атрибуты объектов соединений.

c.row_factory

Функция, которая вызывается для создания объекта, представляющего содержимое записи в возвращаемом наборе данных. Эта функция должна принимать два аргумента: объект курсора, используемый для получения результатов, и кортеж с исходной записью.

c.text_factory

Функция, которая вызывается для создания объекта, представляющего текстовые значения в базе данных. Функция должна принимать единственный аргумент – строку байтов в кодировке UTF-8. Возвращаемое значение должно быть строкой какого-либо типа. По умолчанию возвращается строка Юникода.

c.total_changes

Целое число записей, которые были изменены с момента открытия соединения с базой данных.

Последняя особенность объектов соединений заключается в том, что они могут использоваться совместно с менеджером контекста, обеспечивая автоматическую обработку транзакций. Например:

```
conn = sqlite.connect("somedb")
with conn:
    conn.execute("insert into sometable values (?,?)", ("foo","bar")))
```

В данном примере после выполнения всех инструкций в блоке with, если не возникло никаких ошибок, автоматически будет вызван метод commit(). Если будет возбуждено какое-либо исключение, будет выполнена операция rollback(), а исключение будет возбуждено повторно.

Курсоры и основные операции

Прежде чем приступать к выполнению операций в базе данных sqlite3, необходимо создать объект курсора, с помощью метода cursor() объекта соединения. После этого можно будет использовать методы курсора execute(), executemany() и executescript() для выполнения инструкций SQL. Общее описание этих методов приводится в разделе «Прикладной интерфейс доступа к реляционным базам данных». Вместо того чтобы повторять эту информацию здесь, мы рассмотрим примеры типичных случаев их использования вместе с примерами программного кода. Цель этого раздела состоит в том, чтобы показать, как действуют объекты курсоров, и продемонстрировать некоторые наиболее типичные запросы SQL – для тех, кому необходимо быстро вспомнить их синтаксис.

Создание новых таблиц в базе данных

В следующем примере показано, как открывать базу данных и как создавать новые таблицы:

```
import sqlite3
conn = sqlite3.connect("mydb")
cur = conn.cursor()
cur.execute("create table stocks (symbol text, shares integer, price real)")
conn.commit()
```

При определении новых таблиц необходимо использовать элементарные типы данных SQLite: `text`, `integer`, `real` и `blob`. Тип `blob` представляет строки байтов, тогда как тип `text` представляет строки Юникода в кодировке UTF-8.

Добавление новых значений в таблицы

Следующий пример демонстрирует, как выполняется добавление новых записей в таблицу:

```
import sqlite3
conn = sqlite3.connect("mydb")
cur = conn.cursor()
cur.execute("insert into stocks values (?,?,?)",('IBM',50,91.10))
cur.execute("insert into stocks values (?,?,?)",('AAPL',100,123.45))
conn.commit()
```

При добавлении новых значений всегда желательно использовать символы подстановки ?, как показано в примере. Каждый символ ? замещается значением из кортежа, который передается в качестве параметра.

Для добавления в таблицу целых последовательностей данных можно использовать метод курсора `executemany()`, как показано ниже:

```
stocks = [ ('GOOG',75,380.13),
           ('AA',60,14.20),
           ('AIG',125, 0.99) ]
cur.executemany("insert into stocks values (?,?,?)",stocks)
```

Изменение существующих записей

Ниже приводится пример, как можно изменять значения столбцов в существующих записях:

```
cur.execute("update stocks set shares=? where symbol=?",(50,'IBM'))
```

Как и при добавлении новых строк, здесь также желательно использовать в инструкциях SQL символы подстановки ? и передавать фактические значения в виде кортежа.

Удаление записей

Следующий пример демонстрирует, как удалять записи:

```
cur.execute("delete from stocks where symbol=?",('SCOX',))
```

Выполнение простых запросов

Следующий пример демонстрирует, как можно выполнять простые запросы и получать результаты:

```
# Выбрать все данные из таблицы
for row in cur.execute("select * from stocks"):
    инструкции

# Выбрать только некоторые столбцы
for shares, price in cur.execute("select shares,price from stocks"):
    инструкции

# Выбрать записи, соответствующие условию
for row in cur.execute("select * from stocks where symbol=?",('IBM',))
    инструкции

# Выбрать записи, соответствующие условию, и отсортировать их
for row in cur.execute("select * from stocks order by shares"):
    инструкции

# Выбрать записи, соответствующие условию,
# и отсортировать их в обратном порядке
for row in cur.execute("select * from stocks order by shares desc"):
    инструкции

# Выполнить соединение таблиц по общему столбцу (symbol)
for row in cur.execute("""select s.symbol, s.shares, p.price
                          from stocks as s, prices as p using(symbol)"""):
    инструкции
```

Модули доступа к базам данных типа DBM

В состав стандартной библиотеки языка Python входят несколько модулей, обеспечивающих доступ к файлам баз данных типа UNIX DBM. Поддерживается несколько стандартных типов таких баз данных. Для чтения стандартных файлов баз данных UNIX DBM используется модуль `dbm`. Модуль `gdbm` используется для чтения файлов баз данных GNU dbm (*http://www.gnu.org/software/gdbm*). Модуль `dbhash` используется для чтения файлов баз данных, созданных с помощью библиотеки Berkeley DB (*http://www.oracle.com/database/berkeley-db/index.html*). Модуль `dumbdbm`, написанный исключительно на языке Python, реализует свою собственную простейшую базу данных типа DBM.

Все эти модули предоставляют объекты, которые реализуют доступ к хранимым данным по образу и подобию словарей, основанных на строках. То есть они действуют, как обычные словари Python, с тем ограничением, что все ключи и значения могут быть только строками. Файл базы данных обычно открывается с помощью версии функции `open()`.

`open(`*`filename`* `[, ` *`flag`* `[, ` *`mode`*`]])`

Эта функция открывает файл базы данных с именем *`filename`* и возвращает объект базы данных. В аргументе *`flag`* передается значение `'r'` – для досту-

па к базе данных только для чтения, 'w' – для доступа на чтение и на запись, 'c' – для создания файла базы данных, если он отсутствует, или 'n', чтобы принудительно создать новый файл базы данных. В аргументе *mode* передается целое число, определяющее, какой режим доступа будет установлен при создании нового файла базы данных (в UNIX по умолчанию используется значение 0666).

Объект, возвращаемый функцией open(), как минимум поддерживает следующие операции над словарями:

Операция	Описание
d[*key*] = *value*	Вставляет новое значение *value* в базу данных
value = *d*[*key*]	Извлекает значение из базы данных
del *d*[*key*]	Удаляет элемент данных из базы
d.close()	Закрывает базу данных
key in *d*	Проверяет наличие ключа *key* в базе данных
d.sync()	Сохраняет все изменения в файле

Каждая конкретная реализация может добавлять свои особенности (за дополнительной информацией о них обращайтесь к справочным руководствам соответствующих модулей).

Одна из проблем, связанных с модулями доступа к базам данных типа DBM, заключается в том, что не все модули могут быть установлены на разные платформы. Например, модули dbm и gdbm не поддерживают операционную систему Windows. Однако программы на языке Python все-таки могут создавать базы данных типа DBM для своего использования. Для решения этой проблемы в состав Python входит модуль anydbm, который способен открывать и создавать файлы базы данных DBM. Этот модуль предоставляет функцию open(), описанную выше, и гарантирует работоспособность во всех платформах. Это достигается за счет поиска доступных модулей DBM и выбора наиболее мощной библиотеки (обычно выбирается библиотека dbhash, если она доступна). В крайнем случае используется модуль dumbdbm, который доступен всегда.

Также существует еще один модуль whichdb, который содержит функцию whichdb(*filename*). Эта функция может использоваться для проверки файла с целью определить тип базы данных DBM, с помощью которой он был создан.

В приложениях, для которых проблема переносимости имеет большое значение, как правило, лучше не полагаться на использование этих низкоуровневых модулей. Например, если базу данных DBM создать в одной операционной системе, а затем перенести ее в другую операционную систему, есть вероятность, что программа на языке Python не сможет прочитать его, если в системе не будет установлен соответствующий модуль DBM. Не следует также использовать эти модули для организации хранения больших

объемов данных или в ситуациях, когда несколько программ на языке Python могут одновременно открывать один и тот же файл базы данных, или когда требуется высокая надежность и поддержка транзакций (в подобных случаях более удачным выбором будет модуль sqlite3).

Модуль shelve

Модуль shelve обеспечивает поддержку возможности сохранения объектов, используя для этого специальный объект «хранилища». Своим поведением этот объект напоминает словарь, за исключением того, что все объекты, которые он содержит, сохраняются в базе данных на основе хеш-таблицы, такой как dbhash, dbm или gdbm. Однако, в отличие от этих модулей, круг сохраняемых объектов не ограничивается строками. Сохранить можно любой объект, который совместим с модулем pickle. Хранилище **создается вызовом функции** shelve.open().

open(*filename* [,*flag*='c' [, *protocol* [, *writeback*]]])

Открывает файл хранилища. Создает новый файл, если он не существует. В аргументе *filename* передается имя файла базы данных без расширения. Аргумент *flag* имеет тот же смысл, как описанный в начале раздела «Модули доступа к базам данных типа DBM», и может иметь одно из значений: 'r', 'w', 'c' или 'n'. Аргумент *protocol* определяет, какой протокол будет использоваться для сериализации объектов, сохраняемых в базе данных. Он имеет то же назначение, что и в модуле pickle. Аргумент *writeback* определяет политику кэширования объекта базы данных. Если он имеет значение True, все элементы, к которым выполнялось обращение, кэшируются в памяти и сохраняются в файле только в момент закрытия хранилища. По умолчанию используется значение False. Возвращает объект хранилища.

После открытия хранилища над ним могут выполняться следующие операции:

Операция	Описание
d[*key*] = *data*	Сохраняет данные с ключом *key*. Затирает существующие данные.
data = *d*[*key*]	Извлекает данные с ключом *key*.
del *d*[*key*]	Удаляет данные с ключом *key*.
d.has_key(*key*)	Проверяет наличие ключа *key* в хранилище.
d.keys()	Возвращает все ключи.
d.close()	Закрывает хранилище.
d.sync()	Сохраняет все изменения на диске.

В качестве ключей в хранилище могут использоваться только строки. Объекты, сохраняемые в хранилище, должны поддерживать возможность сериализации с помощью модуля pickle.

`Shelf(`*`dict`* `[, `*`protocol`* `[, `*`writeback`*`]])`

Класс-примесь, который реализует функциональность хранилища поверх объекта словаря *dict*. Когда объекты сохраняются в возвращаемом объекте хранилища, они сериализуются и запоминаются в словаре *dict*. Аргументы *protocol* и *writeback* имеют тот же смысл, что и в функции `shelve.open()`.

Модуль `shelve` использует модуль `anydbm` для выбора соответствующего модуля DBM. В большинстве стандартных конфигураций Python чаще всего выбирается модуль `dbhash`, который основан на использовании библиотеки Berkeley DB.

18

Работа с файлами и каталогами

В этой главе описываются модули Python, предназначенные для работы с файлами и каталогами на высоком уровне. В число рассматриваемых входят модули для работы с основными форматами сжатия файлов, такими как gzip и bzip2, модули для извлечения файлов из архивов, таких как zip и tar, и модули для манипулирования самой файловой системой (например, получение содержимого каталогов, перемещение, переименование, копирование и так далее). Низкоуровневые средства для работы с файлами рассматриваются в главе 19 «Службы операционной системы». Модули, позволяющие реализовать синтаксический анализ содержимого таких файлов, как XML и HTML, главным образом рассматриваются в главе 24 «Обработка и кодирование данных в Интернете».

Модуль bz2

Модуль `bz2` используется для чтения и записи данных, сжатых по алгоритму bzip2.

`BZ2File(`*`filename`* `[, ` *`mode`* `[, ` *`buffering`* `[, ` *`compresslevel`*`]]])`

Открывает сжатый файл `.bz2` с именем *`filename`* и возвращает объект, похожий на объект файла. В аргументе *`mode`* передается значение `'r'`, когда файл открывается для чтения, и `'w'`, когда файл открывается для записи. Кроме того, доступна поддержка универсального символа перевода строки, для чего в аргументе *`mode`* следует указать значение `'rU'`. В аргументе *`buffering`* определяется размер буфера в байтах. По умолчанию используется значение `0` (буферизация отключена). В аргументе *`compresslevel`* передается число от 1 до 9. Значение 9 (по умолчанию) соответствует наивысшей степени сжатия, но при этом на обработку файла уходит больше времени. Возвращаемый объект поддерживает все операции, обычные для файлов, включая `close()`, `read()`, `readline()`, `readlines()`, `seek()`, `tell()`, `write()` и `writelines()`.

`BZ2Compressor([compresslevel])`

Создает объект, выполняющий сжатие последовательностей блоков данных. Аргумент *compresslevel* определяет степень сжатия, которая выражается числом от 1 до 9 (по умолчанию).

Экземпляр *c* класса `BZ2Compressor` обладает следующими двумя методами:

`c.compress(data)`

Передает новую строку объекту сжатия *c*. Возвращает строку со сжатыми данными, если это возможно. Поскольку сжатие данных выполняется порциями, возвращаемая строка может включать не все исходные данные или включать сжатые данные, полученные в результате предыдущих вызовов метода `compress()`. Чтобы получить все оставшиеся данные, хранящиеся в объекте сжатия, после того, как ему будут переданы последние исходные данные, следует вызвать метод `flush()`.

`c.flush()`

Выталкивает внутренние буферы и возвращает строку, содержащую сжатую версию всех оставшихся данных. После выполнения этой операции метод `compress()` объекта больше не должен вызываться.

`BZ2Decompressor()`

Создает объект, выполняющий распаковывание сжатых данных.

Экземпляр *d* класса `BZ2Decompressor` поддерживает всего один метод:

`d.decompress(data)`

Получив блок сжатых данных в виде строки *data*, этот метод возвращает распакованные данные. Поскольку обработка данных выполняется порциями, возвращаемая строка может включать не все распакованные данные. Повторные вызовы этого метода будут продолжать распаковывать блоки данных, пока не встретится признак конца потока. После этого все последующие попытки распаковать данные будут приводить к исключению `EOFError`.

`compress(data [, compresslevel])`

Возвращает сжатую версию исходных данных в строке *data*. В аргументе *compresslevel* передается число в диапазоне от 1 до 9 (по умолчанию).

`decompress(data)`

Возвращает строку с распакованными данными, содержащимися в строке *data*.

Модуль filecmp

Модуль `filecmp` предоставляет функции, которые могут использоваться для сравнивания файлов и каталогов:

`cmp(file1, file2 [, shallow])`

Сравнивает файлы *file1* и *file2* и возвращает `True`, если они равны, в противном случае возвращает `False`. По умолчанию равными считаются фай-

лы, для которых функция os.stat() возвращает одинаковые значения атрибутов. Если в аргументе *shallow* функции передается значение False, дополнительно сравнивается содержимое файлов.

cmpfiles(*dir1*, *dir2*, *common* [, *shallow*])

Сравнивает файлы, перечисленные в списке *common*, в двух каталогах *dir1* и *dir2*. Возвращает кортеж с тремя списками имен файлов (*match*, *mismatch*, *errors*). В списке *match* перечислены одинаковые файлы, в списке *mismatch* – отличающиеся файлы и в списке *errors* – файлы, сравнивание которых не может быть выполнено по каким-либо причинам. Аргумент *shallow* имеет тот же смысл, что и в функции cmp().

dircmp(*dir1*, *dir2* [, *ignore* [, *hide*]])

Создает объект сравнивания каталогов, который может использоваться для выполнения различных операций сравнивания каталогов *dir1* и *dir2*. В аргументе *ignore* передается список имен, которые следует исключить из операции сравнивания, и который по умолчанию содержит значения ['RCS','CVS','tags']. В аргументе *hide* передается список имен для сокрытия; он по умолчанию содержит значения [os.curdir, os.pardir] ([‘.’, ‘..’] в UNIX).

Объект сравнивания каталогов *d*, возвращаемый функцией dircmp(), обладает следующими методами и атрибутами:

d.report()

Сравнивает каталоги *dir1* и *dir2* и выводит отчет в поток sys.stdout.

d.report_partial_closure()

Сравнивает каталоги *dir1* и *dir2*, а также общие подкаталоги следующего уровня. Результаты выводятся в поток sys.stdout.

d.report_full_closure()

Сравнивает каталоги *dir1* и *dir2*, а также все вложенные подкаталоги, рекурсивно. Результаты выводятся в поток sys.stdout.

d.left_list

Список всех файлов и подкаталогов в каталоге *dir1*. Содержимое списка фильтруется в соответствии с содержимым списков *hide* и *ignore*.

d.right_list

Список всех файлов и подкаталогов в каталоге *dir2*. Содержимое списка фильтруется в соответствии с содержимым списков *hide* и *ignore*.

d.common

Список всех файлов и подкаталогов, найденных в обоих каталогах *dir1* и *dir2*.

d.left_only

Список всех файлов и подкаталогов, найденных в каталоге *dir1*.

d.right_only

Список всех файлов и подкаталогов, найденных в каталоге *dir2*.

d.common_dirs

Список подкаталогов, общих для каталогов *dir1* и *dir2*.

d.common_files

Список файлов, общих для каталогов *dir1* и *dir2*.

d.common_funny

Список файлов в каталогах *dir1* и *dir2* с одинаковыми именами, которые имеют разные типы или для которых невозможно получить информацию с помощью функции os.stat().

d.same_files

Список файлов с идентичным содержимым в каталогах *dir1* и *dir2*.

d.diff_files

Список файлов с разным содержимым в каталогах *dir1* и *dir2*.

d.funny_files

Список файлов, присутствующих в обоих каталогах *dir1* и *dir2*, сравнивание которых не может быть выполнено по каким-либо причинам (например, из-за отсутствия прав доступа).

d.subdirs

Словарь, отображающий имена в *d*.common_dirs в дополнительные объекты dircmp.

Примечание

Значения атрибутов объекта dircmp вычисляются в момент обращения к ним, а не в момент создания объекта. Благодаря этому, когда интерес представляет значение лишь некоторых из атрибутов, наличие неиспользуемых атрибутов не оказывает отрицательного влияния на производительность.

Модуль fnmatch

Модуль fnmatch обеспечивает поддержку сопоставления имен файлов с применением шаблонных символов, как это делает командная оболочка в системе UNIX. Этот модуль может использоваться только для сопоставления имен файлов с шаблоном, а получить фактический список файлов, соответствующих шаблону, можно с помощью модуля glob. Шаблоны имеют следующий синтаксис:

Символ(ы)	Описание
*	Соответствует любому количеству любых символов
?	Соответствует одному любому символу
[*seq*]	Соответствует любому символу из множества *seq*
[!*seq*]	Соответствует любому символу, не входящему в множество *seq*

Для сопоставления имен файлов с шаблонами могут использоваться следующие функции:

`fnmatch(`*`filename`*`, `*`pattern`*`)`

Возвращает `True` или `False`, в зависимости от того, соответствует ли имя файла *`filename`* шаблону *`pattern`*. Чувствительность к регистру символов зависит от операционной системы (в некоторых системах, таких как Windows, регистр символов не учитывается).

`fnmatchcase(`*`filename`*`, `*`pattern`*`)`

Выполняет сопоставление имени файла *`filename`* с шаблоном *`pattern`* с учетом регистра символов.

`filter(`*`names`*`, `*`pattern`*`)`

Применяет функцию `fnmatch()` ко всем именам файлов, перечисленных в последовательности *`names`*, и возвращает список всех имен, соответствующих шаблону *`pattern`*.

Примеры

```
fnmatch('foo.gif', '*.gif')               # Вернет True
fnmatch('part37.html', 'part3[0-5].html') # Вернет False

# Пример поиска файлов в дереве каталогов
# с помощью функции os.walk(), модуля fnmatch и генераторов
def findall(topdir, pattern):
    for path, files, dirs in os.walk(topdir):
        for name in files:
            if fnmatch.fnmatch(name,pattern):
                yield os.path.join(path,name)

# Отыскать все файлы с расширением .py
for pyfile in findall(".","*.py"):
    print pyfile
```

Модуль glob

Модуль `glob` позволяет получить список всех имен файлов в каталоге, соответствующих указанному шаблону, с применением правил командной оболочки UNIX (которые были представлены в описании модуля `fnmatch`).

`glob(`*`pattern`*`)`

Возвращает список путей к файлам, соответствующих шаблону *`pattern`*.

`iglob(`*`pattern`*`)`

Возвращает тот же результат, что и функция `glob()`, но в виде итератора.

Пример

```
htmlfile = glob('*.html')
imgfiles = glob('image[0-5]*.gif')
```

Примечание

Знак тильды (~) не имеет специального значения, как в командной оболочке; а также не выполняется подстановка значений переменных окружения. Если это необходимо, используйте функции `os.path.expanduser()` и `os.path.expandvars()` соответственно, перед вызовом функции `glob()`.

Модуль gzip

Модуль `gzip` объявляет класс `GzipFile`, который может использоваться для чтения и записи в файлы, совместимые с утилитой GNU `gzip`. Объекты класса `GzipFile` действуют как обычные объекты файлов, за исключением того, что при этом данные автоматически сжимаются и распаковываются.

`GzipFile([`*`filename`* `[, `*`mode`* `[, `*`compresslevel`* `[, `*`fileobj`*`]]]])`

Создает объект класса `GzipFile` и открывает файл. В аргументе *`filename`* передается имя файла, а в аргументе *`mode`* – одно из значений: '`r`', '`rb`', '`a`', '`ab`', '`w`' или '`wb`'. По умолчанию используется режим '`rb`'. В аргументе *`compresslevel`* передается целое число от 1 до 9, которое управляет степенью сжатия. Значение `1` обеспечивает самую низкую степень сжатия и самую высокую скорость работы; значение `9` (используется по умолчанию) – самую высокую степень сжатия и самую низкую скорость работы. В аргументе *`fileobj`* передается существующий объект файла. Если этот аргумент указан при вызове функции, его значение будет использоваться вместо имени файла *`filename`*.

`open(`*`filename`* `[, `*`mode`* `[, `*`compresslevel`*`]])`

Аналогична вызову функции `GzipFile(`*`filename, mode, compresslevel`*`)`. По умолчанию аргумент *`mode`* получает значение '`rb`', а аргумент *`compresslevel`* – значение 9.

Примечания

- Вызов метода `close()` объекта класса `GzipFile` не закрывает файл, переданный в аргументе *`fileobj`*. Это позволяет записать в файл дополнительную информацию вслед за сжатыми данными.
- Файлы, созданные утилитой UNIX `compress`, не поддерживаются.
- Этот модуль использует модуль `zlib`.

Модуль shutil

Модуль `shutil` используется для выполнения таких операций высокого уровня над файлами, как копирование, удаление и переименование. Функции из этого модуля должны применяться только к обычным файлам и каталогам. В частности, они не могут работать со специальными файлами, такими как именованные каналы, блочные устройства и так далее. Кроме того, имейте в виду, что эти функции не всегда корректно обрабатывают некоторые дополнительные типы метаданных (например, ответвления ресурсов, коды создателей и другие).

`copy(src,dst)`

Копирует файл *src* в файл или каталог *dst*, с сохранением прав доступа. Значения аргументов *src* и *dst* должны быть строками.

`copy2(src, dst)`

Действует аналогично функции `copy()`, но дополнительно копирует время последнего обращения и время последнего изменения.

`copyfile(src, dst)`

Копирует содержимое файла *src* в файл *dst*. Значения аргументов *src* и *dst* должны быть строками.

`copyfileobj(f1, f2 [, length])`

Копирует все данные из открытого объекта файла *f1* в открытый объект файла *f2*. Аргумент *length* определяет максимальный размер буфера. Отрицательное значение в аргументе *length* означает, что копирование данных будет выполнено целиком, одной операцией (то есть все данные будут прочитаны как единый блок, а затем записаны).

`copymode(src, dst)`

Копирует биты разрешений из файла *src* в файл *dst*.

`copystat(src, dst)`

Копирует биты разрешений, время последнего обращения и время последнего изменения из файла *src* в файл *dst*. Содержимое файла *dst*, владелец и группа остаются без изменений.

`copytree(src, dst, symlinks [,ignore]])`

Рекурсивно копирует дерево каталогов с корнем в каталоге *src*. Каталог назначения *dst* не должен существовать (он будет создан). Копирование файлов выполняется с помощью функции `copy2()`. Если в аргументе *symlinks* передается истинное значение, символические ссылки в исходном дереве каталогов будут представлены символическими ссылками в новом дереве. Если в аргументе *symlinks* передается ложное значение или он опущен, в новое дерево каталогов будет скопировано содержимое файлов, на которые указывают символические ссылки. В необязательном аргументе *ignore* передается функция, которая будет использоваться для фильтрации файлов. Эта функция должна принимать имя каталога и список его содержимого и возвращать список имен файлов, которые не должны копироваться. Если в процессе копирования будут происходить какие-либо ошибки, они будут собраны все вместе и в конце будет возбуждено исключение `Error`. В качестве аргумента исключению будет передан список кортежей (*srcname, dstname, exception*), по одному для каждой возникшей ошибки.

`ignore_patterns(pattern1, pattern2, ...)`

Создает функцию, которая может использоваться для исключения из операции имен, соответствующих шаблонам *pattern1*, *pattern2* и так далее. Возвращаемая функция принимает два аргумента. В первом она принимает имя каталога, а во втором – список содержимого этого каталога. В качестве результата она возвращает список имен файлов, которые должны быть ис-

ключены из операции. Обычно эта функция используется, как значение аргумента *ignore* при вызове функции `copytree()`. Однако она также может использоваться в операциях с привлечением функции `os.walk()`.

`move(`*src*`, `*dst*`)`

Перемещает файл или каталог *src* в *dst*. Если *src* – это каталог и он перемещается в другую файловую систему, выполняется рекурсивное копирование его содержимого.

`rmtree(`*path* `[, `*ignore_errors* `[, `*onerror*`]])`

Удаляет дерево каталогов целиком. Если в аргументе *ignore_errors* передается истинное значение, ошибки, возникающие при удалении, будут игнорироваться. В противном случае для обработки ошибок будет вызываться функция, переданная в аргументе *onerror*. Эта функция должна принимать три аргумента (*func*, *path* и *excinfo*), где *func* – функция, которая вызвала ошибку (`os.remove()` или `os.rmdir()`), *path* – путь, который был передан функции, и *excinfo* – информация об исключении, полученная вызовом функции `sys.exc_info()`. Если аргумент *onerror* опущен, появление ошибок будет вызывать исключение.

Модуль tarfile

Модуль `tarfile` используется для работы с файлами архивов tar. С помощью функций из этого модуля можно читать и записывать как сжатые, так и несжатые файлы tar.

`is_tarfile(`*name*`)`

Возвращает `True`, если *name* соответствует допустимому файлу tar, который может быть прочитан с помощью этого модуля.

`open([`*name* `[, `*mode* `[, `*fileobj* `[, `*bufsize*`]]]])`

Создает новый объект класса `TarFile`. В аргументе *name* передается путь к файлу, а в аргументе *mode* – строка, определяющая режим открытия файла tar. Строка режима в аргументе *mode* является комбинацией режима доступа к файлу и схемы сжатия и имеет вид '*filemode*[:*compression*]'. В число допустимых входят следующие комбинации:

Режим	Описание
'r'	Файл открывается для чтения. Если файл сжат, он будет автоматически распакован. Этот режим используется по умолчанию.
'r:'	Файл открывается для чтения без сжатия.
'r:gz'	Файл открывается для чтения со сжатием gzip.
'r:bz2'	Файл открывается для чтения со сжатием bzip2.
'a', 'a:'	Файл открывается для добавления в конец без сжатия.
'w', 'w:'	Файл открывается для записи без сжатия.
'w:gz'	Файл открывается для записи со сжатием gzip.
'w:bz2'	Файл открывается для записи со сжатием bzip2.

Для создания объектов класса TarFile, допускающих только последовательный доступ (не допускается произвольное перемещение по файлу), используются следующие режимы:

Режим	Описание
'r|'	Открывает поток несжатых блоков для чтения
'r|gz'	Открывает поток со сжатием gzip для чтения
'r|bz2'	Открывает поток со сжатием bzip2 для чтения
'w|'	Открывает поток несжатых блоков для записи
'w|gz'	Открывает поток со сжатием gzip для записи
'w|bz2'	Открывает поток со сжатием bzip2 для записи

Если указан аргумент *fileobj*, он должен быть открытым объектом файла. В этом случае вместо аргумента *name* будет использоваться аргумент *fileobj*. Аргумент *bufsize* определяет размер блока в файле tar. По умолчанию используется размер блока 20×512 байтов.

Экземпляр *t* класса TarFile, возвращаемый функцией open(), поддерживает следующие методы и атрибуты:

t.add(*name* [, *arcname* [, *recursive*]])

Добавляет новый файл в архив tar. В аргументе *name* передается имя файла любого типа (каталога, символической ссылки и так далее). Аргумент *arcname* определяет альтернативное имя файла внутри архива. В аргументе *recursive* передается логическое значение, определяющее, должно ли добавляться содержимое каталогов рекурсивно. По умолчанию имеет значение True.

t.addfile(*tarinfo* [, *fileobj*])

Добавляет новый объект файла в архив tar. В аргументе *tarinfo* передается экземпляр структуры TarInfo с информацией об элементе архива. В аргументе *fileobj* передается открытый объект файла, из которого будут читаться данные и сохраняться в архиве. Объем данных для чтения определяется из атрибута size аргумента *tarinfo*.

t.close()

Закрывает архив tar и записывает в конец архива два пустых блока, если он был открыт для записи.

t.debug

Определяет, какой объемом отладочной информации будет выводиться. Если имеет значение 0, никакой информации выводиться не будет, а при значении 3 будут выводиться все отладочные сообщения. Вывод сообщений производится в поток sys.stderr.

t.dereference

Если этот атрибут установлен в значение True, символические и жесткие ссылки разыменовываются и в архив добавляется содержимое файлов, на

которые указывают эти ссылки. Если он установлен в значение `False`, добавляются только ссылки.

t.`errorlevel`

Определяет политику обработки ошибок, возникающих при извлечении элементов архива. Если этот атрибут установлен в значение 0, ошибки будут игнорироваться. Если он установлен в значение 1, ошибки будут приводить к возбуждению исключения `OSError` или `IOError`. Если он установлен в значение 2, нефатальные ошибки будут приводить к возбуждению исключения `TarError`.

t.`extract(`*member* `[, ` *path*`])`

Извлекает элемент *member* из архива и сохраняет его в текущем каталоге. В аргументе *member* может передаваться либо имя элемента архива, либо экземпляр класса `TarInfo`. Аргумент *path* может использоваться, чтобы указать иной каталог для сохранения.

t.`extractfile(`*member*`)`

Извлекает элемент *member* из архива и возвращает объект файла, доступный только для чтения, который может использоваться для чтения содержимого с помощью методов `read()`, `readline()`, `readlines()`, `seek()` и `tell()`. В аргументе *member* может передаваться либо имя элемента архива, либо экземпляр класса `TarInfo`. Если элемент *member* представляет ссылку, выполняется попытка открыть файл, на который указывает ссылка.

t.`getmember(`*name*`)`

Отыскивает в архиве элемент с именем *name* и возвращает объект класса `TarInfo` с информацией о нем. Если искомый элемент в архиве отсутствует, возбуждается исключение `KeyError`. Если в архиве имеется несколько элементов с именем *name*, возвращается информация о последнем элементе (который, как предполагается, является наиболее свежим).

t.`getmembers()`

Возвращает список объектов класса `TarInfo` для всех элементов архива.

t.`getnames()`

Возвращает список имен всех элементов архива.

t.`gettarinfo([`*name* `[, ` *arcname* `[, ` *fileobj*`]]])`

Возвращает объект класса `TarInfo`, соответствующий файлу с именем *name* в файловой системе или открытому объекту файла *fileobj*. В аргументе *arcname* передается альтернативное имя объекта в архиве. Основное назначение этой функции — создание объектов класса `TarInfo` для использования в вызовах таких методов, как `add()`.

t.`ignore_zeros`

Если этот атрибут установлен в значение `True`, при чтении архива пустые блоки будут пропускаться. Если он установлен в значение `False` (по умолчанию), пустой блок будет интерпретироваться, как признак конца архива. Установка этого атрибута в значение `True` может пригодиться при чтении поврежденных архивов.

t.list([*verbose*])

Выводит перечень содержимого архива в поток sys.stdout. Аргумент *verbose* определяет степень подробности выводимой информации. Если в этот аргумент передать значение False, будут выводиться только имена элементов архива. В противном случае будет выводиться подробный отчет (по умолчанию).

t.next()

Используется при выполнении итераций по элементам архива. Возвращает структуру TarInfo для следующего элемента архива или None.

t.posix

Если этот атрибут установлен в значение True, tar-файл создается в соответствии с требованиями стандарта POSIX 1003.1-1990. Этот стандарт накладывает ограничения на длину имен файлов и на размеры самих файлов (имена файлов не могут превышать 256 символов, а размер файлов не может превышать 8 Гбайт). Если этот атрибут установлен в значение False, архив создается в соответствии с дополнениями GNU к стандартам, которые ослабляют эти ограничения. По умолчанию используется значение False.

Многие из предыдущих методов оперируют экземплярами класса TarInfo. В следующей таблице перечислены методы и атрибуты экземпляра *ti* класса TarInfo.

Атрибут	Описание
ti.gid	Числовой идентификатор группы
ti.gname	Имя группы
ti.isblk()	Возвращает True, если объект является блочным устройством
ti.ischr()	Возвращает True, если объект является символьным устройством
ti.isdev()	Возвращает True, если объект является устройством (символьным, блочным или каналом FIFO)
ti.isdir()	Возвращает True, если объект является каталогом
ti.isfifo()	Возвращает True, если объект является каналом FIFO
ti.isfile()	Возвращает True, если объект является обычным файлом
ti.islnk()	Возвращает True, если объект является жесткой ссылкой
ti.isreg()	То же, что и метод isfile()
ti.issym()	Возвращает True, если объект является символической ссылкой
ti.islinkname	Имя файла, на который ссылается жесткая или символическая ссылка
ti.mode	Биты разрешений
ti.mtime	Время последнего изменения
ti.name	Имя элемента архива
ti.size	Размер в байтах

(продолжение)

Атрибут	Описание
ti.type	Тип файла, может быть одним из следующих: REGTYPE, AREGTYPE, LNKTYPE, SYMTYPE, DIRTYPE, FIFOTYPE, CONTTYPE, CHRTYPE, BLKTYPE или GNUTYPE_SPARSE
ti.uid	Числовой идентификатор пользователя
ti.uname	Имя пользователя

Исключения

В модуле tarfile объявлены следующие исключения:

`TarError`

Базовый класс всех остальных исключений.

`ReadError`

Возбуждается в случае ошибки открытия tar-файла (например, при попытке открыть файл, не являющийся архивом tar).

`CompressionError`

Возбуждается, когда сжатые данные не могут быть распакованы.

`StreamError`

Возбуждается при попытке выполнить операцию, не поддерживаемую объектами класса TarFile (например, операцию, для выполнения которой требуется произвольный доступ к разным частям файла).

`ExtractError`

Возбуждается при появлении нефатальных ошибок в процессе извлечения (только если атрибут errorlevel установлен в значение 2).

Пример

```
# Открыть архив tar и поместить в него несколько файлов
t = tarfile.open("foo.tar","w")
t.add("README")
import glob
for pyfile in glob.glob("*.py"):
    t.add(pyfile)
t.close()

# Открыть архив tar и выполнить обход всех его элементов
t = tarfile.open("foo.tar")
for f in t:
    print("%s %d" % (f.name, f.size))

# Просмотреть содержимое архива tar
# и вывести содержимое файлов с именем "README"
t = tarfile.open("foo.tar")
for f in t:
```

```
if os.path.basename(f.name) == "README":
    data = t.extractfile(f).read()
    print("**** %s ****" % f.name)
```

Модуль tempfile

Модуль `tempfile` используется для создания временных файлов.

`mkdtemp([`*`suffix`* `[,`*`prefix`* `[, `*`dir`*`]]])`

Создает временный каталог, доступный только владельцу вызывающего процесса, и возвращает абсолютный путь к этому каталогу. В аргументе *suffix* можно указать окончание, которое будет добавлено в конец сгенерированного имени каталога, в аргументе *prefix* – префикс, который будет добавлен в начало имени каталога, и в аргументе *dir* – путь к каталогу, где должен быть создан временный каталог.

`mkstemp([`*`suffix`* `[,`*`prefix`* `[, `*`dir`* `[,`*`text`*`]]]])`

Создает временный файл и возвращает кортеж (*fd, pathname*), где *fd* – целочисленный дескриптор файла, возвращаемый функцией `os.open()`, а *pathname* – абсолютный путь к файлу. В аргументе *suffix* можно указать окончание, которое будет добавлено в конец сгенерированного имени файла, в аргументе *prefix* – префикс, который будет добавлен в начало имени файла, в аргументе *dir* – путь к каталогу, где должен быть создан временный файл, и в аргументе *text* – логический флаг, определяющий, в каком режиме должен быть открыт файл – в текстовом или в двоичном (по умолчанию). Гарантируется, что операция создания файла будет атомарной (и безопасной), если операционная система поддерживает флаг `O_EXCL` для функции `os.open()`.

`mktemp([`*`suffix`* `[, `*`prefix`* `[,`*`dir`*`]]])`

Возвращает уникальное имя для временного файла. В аргументе *suffix* можно указать окончание, которое будет добавлено в конец сгенерированного имени файла, в аргументе *prefix* – префикс, который будет добавлен в начало имени файла, в аргументе *dir* – путь к каталогу, где предполагается создать временный файл. Эта функция только генерирует уникальное имя файла и не создает фактический временный файл. Так как при использовании этой функции между созданием имени файла и созданием самого файла проходит некоторое время, появляется опасность, что в течение этого времени другой процесс создаст временный файл с тем же именем. Чтобы избежать этого, лучше использовать функцию `mkstemp()`.

`gettempdir()`

Возвращает имя каталога по умолчанию, где будут создаваться временные файлы.

`gettempprefix()`

Возвращает префикс по умолчанию, который будет использоваться при создании временных файлов. Префикс не включает в себя имя каталога для временных файлов.

`TemporaryFile([mode [, bufsize [, suffix [,prefix [, dir]]]]])`

Создает временный файл вызовом функции `mkstemp()` и возвращает объект, похожий на объект файла, который поддерживает те же методы, что и обычный объект файла. В аргументе *mode* передается режим открытия файла; по умолчанию имеет значение `'w+b'`. Аргумент *bufsize* определяет поведение механизма буферизации и имеет тот же смысл, что и в функции `open()`. Аргументы *suffix*, *prefix* и *dir* имеют тот же смысл, что и в функции `mkstemp()`. Объект, возвращаемый этой функцией, является всего лишь оберткой вокруг обычного объекта файла, доступного в виде атрибута `file`. Файл, созданный этой функцией, автоматически удаляется вместе с уничтожением объекта временного файла.

`NamedTemporaryFile([mode [, bufsize [, suffix [, prefix [, dir [, delete ]]]]]])`

Создает временный файл, как это делает функция `TemporaryFile()`, но гарантирует, что имя файла будет видимо в файловой системе. Имя файла можно получить из атрибута `name` возвращаемого объекта. Обратите внимание, что некоторые системы не позволяют повторно открыть временный файл с тем же именем, пока он не будет закрыт. Если в аргументе *delete* передать значение `True` (по умолчанию), временный файл будет удален сразу же после того, как будет закрыт.

`SpooledTemporaryFile([max_size[, mode[, bufsize[, suffix [, prefix [, dir]]]]]])`

Создает временный файл, как это делает функция `TemporaryFile()`, за исключением того, что содержимое этого файла целиком будет храниться в памяти, пока его размер не превысит значение в аргументе *max_size*. Это достигается за счет того, что первоначально содержимое файла сохраняется в объекте `StringIO`, пока не возникнет необходимость перенести его в файловую систему. При попытке выполнить какую-либо низкоуровневую операцию ввода-вывода над файлом, с привлечением метода `fileno()`, содержимое файла, хранящееся в памяти, будет немедленно записано во временный файл, определенный объектом `TemporaryFile`. Кроме того, объект файла, возвращаемый функцией `SpooledTemporaryFile()`, имеет метод `rollover()`, позволяющий принудительно записать содержимое файла в файловую систему.

При конструировании имен временных файлов используются две глобальные переменные. При необходимости им могут присваиваться новые значения. Значения по умолчанию этих переменных зависят от операционной системы.

Переменная	Описание
`tempdir`	Каталог, в котором будут находиться имена файлов, возвращаемые функцией `mktemp()`.
`template`	Префикс для имен файлов, возвращаемых функцией `mktemp()`. Имя файла формируется из значения переменной `template`, к которому добавляется строка десятичных цифр для придания уникальности.

Примечание

По умолчанию при создании временных файлов модуль tempfile проверяет несколько стандартных местоположений. Например, в системе UNIX временные файлы создаются в одном из каталогов: /tmp, /var/tmp или /usr/tmp. В Windows временные файлы создаются в одном из каталогов: C:\TEMP, C:\TMP, \TEMP или \TMP. Эти каталоги можно поменять, установив значение одной или более переменных окружения TMPDIR, TEMP и TMP. Если по каким-либо причинам временный файл не может быть создан ни в одном из обычных местоположений, он будет создан в текущем рабочем каталоге.

Модуль zipfile

Модуль zipfile используется для работы с архивами в популярном формате zip (ранее известный, как PKZIP; в настоящее время поддерживается широким кругом программ). **Zip-файлы широко используются при работе** с языком Python, в основном для упаковки. Например, если zip-файлы с исходными текстами на языке Python добавить в переменную sys.path, то файлы, содержащиеся в архиве, можно будет загружать с помощью инструкции import (эта возможность реализована в библиотечном модуле zipimport, однако вам едва ли придется использовать его непосредственно). Пакеты, распространяемые в виде файлов с расширением .egg (создаются расширением setuptools), также являются обычными zip-файлами (файл с расширением .egg в действительности является обычным zip-архивом с некоторыми дополнительными метаданными внутри).

Ниже перечислены функции и классы, объявленные в модуле zipfile:

`is_zipfile(`*`filename`*`)`

Проверяет, действительно ли файл *filename* является **zip-файлом. Возвращает** True, если файл *filename* является zip-файлом, и False – в противном случае.

`ZipFile(`*`filename`*` [, `*`mode`*` [, `*`compression`*` [,`*`allowZip64`*`]]])`

Открывает zip-файл *filename* и возвращает экземпляр класса ZipFile. Если в аргументе *mode* передать значение 'r', существующий файл будет открыт для чтения, 'w' – существующий файл будет уничтожен и вместо него будет создан новый файл, 'a' – существующий файл будет открыт в режиме добавления в конец. Для режима 'a', если *filename* соответствует существующему zip-файлу, новые файлы будут добавляться в него. Если файл *filename* не является zip-файлом, архив будет просто добавлен в конец этого файла. Аргумент *compression* определяет метод сжатия, который должен использоваться при записи в архив, и может принимать одно из значений: ZIP_STORED или ZIP_DEFLATED. По умолчанию используется значение ZIP_STORED. **Аргумент** *allowZip64* дает возможность разрешить использование расширений **ZIP64, которые позволяют создавать zip-файлы, размер которых превышает** 2 Гбайта. По умолчанию в этом аргументе передается False.

`PyZipFile(`*`filename`* `[, `*`mode`*`[, `*`compression`* `[,`*`allowZip64`*`]]])`

Открывает zip-файл, как и функция `ZipFile()`, но возвращает экземпляр класса `PyZipFile`, обладающий дополнительным методом `writepy()`, который используется для добавления в архив файлов с исходными текстами на языке Python.

`ZipInfo([`*`filename`* `[, `*`date_time`*`]])`

Создает новый экземпляр класса `ZipInfo` с информацией об элементе архива. Обычно нет необходимости вызывать эту функцию, за исключением случаев использования метода *z*`.writestr()` экземпляра класса `ZipFile` (описывается ниже). В аргументах *`filename`* и *`date_time`* передаются значения для атрибутов `filename` и `date_time`, описываемых ниже.

Экземпляр *z* класса `ZipFile` или `PyZipFile` поддерживает следующие методы и атрибуты:

z`.close()`

Закрывает **файл архива. Должен вызываться перед завершением программы**, чтобы вытолкнуть в zip-файл оставшиеся в памяти записи.

z`.debug`

Степень подробности отладочных сообщений в диапазоне от 0 (сообщения не выводятся) до 3 (выводятся наиболее подробные сообщения).

z`.extract(`*`name`* `[, `*`path`* `[, `*`pwd`* `]])`

Извлекает файл из архива и сохраняет его в текущем рабочем каталоге. В аргументе *`name`* передается строка, полностью описывающая элемент архива, или экземпляр класса `ZipInfo`. Аргумент *`path`* можно использовать, чтобы указать другой каталог, куда должен быть извлечен файл, а в аргументе *`pwd`* можно передать пароль, который будет использоваться при работе с зашифрованными архивами.

z`.extractall([`*`path`* `[`*`members`* `[, `*`pwd`*`]]])`

Извлекает все элементы архива в текущий рабочий каталог. Аргумент *`path`* позволяет определить другой каталог для извлечения, а аргумент *`pwd`* – пароль для работы с зашифрованными архивами. В аргументе *`members`* можно передать список извлекаемых элементов, который должен быть подмножеством списка, возвращаемого методом `namelist()` (описывается ниже).

z`.getinfo(`*`name`*`)`

Возвращает информацию об элементе архива *`name`* в виде экземпляра класса `ZipInfo` (описывается ниже).

z`.infolist()`

Возвращает список объектов класса `ZipInfo` для всех элементов архива.

z`.namelist()`

Возвращает список имен элементов архива.

z`.open(`*`name`* `[, `*`mode`* `[, `*`pwd`*`]])`

Открывает элемент архива с именем *`name`* и возвращает объект, похожий на файл, позволяющий читать содержимое элемента. В аргументе *`name`* **мож-**

но передать строку или экземпляр класса ZipInfo, описывающий один из элементов архива. В аргументе *mode* передается режим открытия файла, который должен быть одним из режимов, обеспечивающих доступ только для чтения, таких как 'r', 'rU' или 'U'. В аргументе *pwd* передается пароль, который будет использоваться для работы с зашифрованными элементами архива. Возвращаемый объект поддерживает методы read(), readline() и readlines(), а также обеспечивает возможность выполнения итераций с помощью инструкции for.

z.printdir()

Выводит каталог архива в поток sys.stdout.

z.read(*name* [,*pwd*])

Читает содержимое элемента *name* архива и возвращает данные в виде строки. В аргументе *name* можно передать строку или экземпляр класса ZipInfo, описывающий один из элементов архива. В аргументе *pwd* передается пароль, который будет использоваться для работы с зашифрованными элементами архива.

z.setpassword(*pwd*)

Устанавливает пароль, который будет использоваться по умолчанию при извлечении зашифрованных файлов из архива.

z.testzip()

Читает все файлы, имеющиеся в архиве, и сверяет контрольные суммы. Возвращает имя первого поврежденного файла или None, если в архиве нет поврежденных файлов.

z.write(*filename*[, *arcname*[, *compress_type*]])

Добавляет файл *filename* в архив под именем элемента архива *arcname*. Аргумент *compress_type* определяет метод сжатия и может иметь значение ZIP_STORED или ZIP_DEFLATED. По умолчанию используется метод сжатия, указанный в вызове функции ZipFile() или PyZipFile(). Архив должен быть открыт в режиме 'w' или 'a'.

z.writepy(*pathname*)

Этот метод доступен только в экземплярах класса PyZipFile и используется для добавления в архив файлов с исходными текстами на языке Python (файлы *.py). Может использоваться как простой способ упаковывания приложений на языке Python для последующего распространения. Если в аргументе *pathname* передается имя файла, это должно быть имя файла .py. В этом случае в архив будет добавлен один из файлов с расширением .pyo, .pyc или .py (в указанном порядке). Если в аргументе *pathname* передается имя каталога, который не является каталогом пакета Python, в архив будут добавлены все файлы с расширениями .pyo, .pyc и .py, имеющиеся в этом каталоге. Если каталог является каталогом пакета, файлы будут добавляться под именем пакета, которое будет использоваться, как путь к файлам. Если какой-либо из подкаталогов также является каталогом пакета, он будет добавлен рекурсивно.

z.writestr(*arcinfo*, *s*)

Записывает строку *s* в zip-файл. В аргументе *arcinfo* может передаваться либо имя файла в архиве, куда будут записаны данные, либо экземпляр класса ZipInfo, содержащий имя файла, дату и время.

Экземпляр *i* класса ZipInfo, возвращаемый функцией ZipInfo() или методами *z*.getinfo() и *z*.infolist(), обладает следующими атрибутами:

Атрибут	Описание
i.filename	Имя элемента архива.
i.date_time	Кортеж (*year*, *month*, *day*, *hours*, *minutes*, *seconds*), содержащий время последнего изменения. Элементы *month* и *day* могут принимать числовые значения в диапазоне 1–12 и 1–31 соответственно. Все остальные значения начинаются с 0.
i.compress_type	Метод сжатия элемента архива. В настоящее время модуль поддерживает только методы ZIP_STORED и ZIP_DEFLATED.
i.comment	Комментарий для элемента архива.
i.extra	Поле данных, содержащее дополнительные атрибуты файла. Какие данные будут храниться в этом поле, зависит от системы, где был создан файл.
i.create_system	Целочисленный код, описывающий систему, где был создан архив. Типичными значениями являются: 0 (MS-DOS FAT), 3 (UNIX), 7 (Macintosh) и 10 (Windows NTFS).
i.create_version	Код версии утилиты PKZIP, с помощью которой был создан архив.
i.extract_version	Минимальная версия утилиты, которая может использоваться для распаковывания архива.
i.reserved	Зарезервированное поле. В настоящее время устанавливается в значение 0.
i.flag_bits	Флаги zip-архива, описывающие представление данных, включая метод шифрования и сжатия.
i.volume	Номер тома в заголовке файла.
i.internal_attr	Описывает внутреннюю структуру содержимого архива. Если самый младший бит имеет значение 1, данные являются текстом ASCII. В противном случае содержимое интерпретируется как двоичные данные.
i.external_attr	Внешние атрибуты файла, количество и назначение которых зависит от операционной системы.
i.header_offset	Смещение в байтах от заголовка файла.
i.file_offset	Смещение в байтах от начала данных в файле.
i.CRC	Контрольная сумма CRC для распакованного файла.
i.compress_size	Размер сжатого файла.
i.file_size	Размер несжатого файла.

Примечание

Подробное описание внутренней структуры **zip-файлов можно найти в примечаниях** к приложению PKZIP, по адресу *http://www.pkware.com/appnote.html*.

Модуль zlib

Модуль `zlib` поддерживает возможность сжатия данных, предоставляя доступ к библиотеке `zlib`.

`adler32(`*`string`* `[,` *`value`*`])`

Вычисляет контрольную сумму Adler-32 строки *`string`*. Аргумент *`value`* используется как начальное значение (например, при вычислении контрольной суммы группы из нескольких строк). В противном случае используется фиксированное значение по умолчанию.

`compress(`*`string`* `[,` *`level`*`])`

Сжимает данные в строке *`string`*. В аргументе *`level`* может передаваться целое число от 1 до 9, определяющее степень сжатия, где 1 соответствует наименьшей степени сжатия (и самой быстрой), а 9 – наивысшей (и самой медленной) степени сжатия. По умолчанию используется значение 6. Возвращает сжатую строку или возбуждает исключение, если возникла какая-либо ошибка.

`compressobj([`*`level`*`])`

Возвращает объект сжатия. Аргумент *`level`* имеет тот же смысл, что и в функции `compress()`.

`crc32(`*`string`* `[,` *`value`*`])`

Вычисляет контрольную сумму CRC строки *`string`*. Значение аргумента *`value`*, если он указан, используется, как начальное значение контрольной суммы. В противном случае используется фиксированное значение.

`decompress(`*`string`* `[,` *`wbits`* `[,` *`buffsize`*`]])`

Распаковывает данные в строке *`string`*. Аргумент *`wbits`* определяет размер буфера окна, а *`buffsize`* – начальный размер выходного буфера. Если возникает какая-либо ошибка, возбуждает исключение.

`decompressobj([`*`wbits`*`])`

Возвращает объект, реализующий распаковывание потока данных. Аргумент *`wbits`* определяет размер буфера окна.

Объект сжатия *`c`* обладает следующими методами:

`c``.compress(`*`string`*`)`

Сжимает строку *`string`*. Возвращает строку со сжатыми данными, при этом может возвращаться только часть сжатых данных из строки *`string`*. Эти данные должны объединяться в общий поток вывода с результатами предыдущих вызовов метода *`c`*`.compress()`. Некоторые входные данные могут сохраняться во внутренних буферах для последующей обработки.

c.flush([*mode*])

Сжимает оставшиеся входные данные и возвращает их в сжатом виде. В аргументе *mode* может передаваться значение Z_SYNC_FLUSH, Z_FULL_FLUSH или Z_FINISH (по умолчанию). Значения Z_SYNC_FLUSH и Z_FULL_FLUSH обеспечивают возможность дальнейшего сжатия и дают возможность частичного восстановления в случае появления ошибок при распаковывании. Значение Z_FINISH завершает поток сжатых данных.

Объект *d*, реализующий распаковывание потока данных, обладает следующими методами и атрибутами:

d.decompress(*string* [,*max_length*])

Распаковывает строку *string* и возвращает строку с распакованными данными, при этом может возвращаться только часть распакованных данных из строки *string*. Эти данные должны объединяться в общий поток вывода с результатами предыдущих вызовов метода *c*.decompress(). Некоторые входные данные могут сохраняться во внутренних буферах для последующей обработки. Аргумент *max_length* определяет максимальный объем блока возвращаемых данных. В случае его превышения еще не обработанные данные помещаются в атрибут *d*.unconsumed_tail.

d.flush()

Распаковывает все оставшиеся входные данные и возвращает их в виде строки. После вызова этого метода объект, реализующий распаковывание потока данных, не может использоваться повторно.

d.unconsumed_tail

Строка, содержащая данные, которые не были обработаны последним вызовом decompress(). В этом атрибуте сохраняются данные, когда распаковывание производится поэтапно, из-за ограничений на размер буфера. Содержимое этого атрибута будет учитываться при последующих вызовах decompress().

d.unused_data

Строка с дополнительными байтами в конце сжатых данных.

Примечание

Библиотеку zlib можно получить по адресу *http://www.zlib.net*.

19

Службы операционной системы

Модули, о которых рассказывается в этой главе, предоставляют доступ к самым разнообразным службам операционной системы, с особым акцентом на низкоуровневые операции ввода-вывода, управление процессами и рабочим окружением. Здесь также рассказывается о модулях, которые обычно используются при разработке системного программного обеспечения, например здесь будут представлены модули, позволяющие читать содержимое файлов с настройками, выполнять запись в файлы журналов и так далее. В главе 18 «Работа с файлами и каталогами» рассказывается о модулях, реализующих высокоуровневые операции над файлами и каталогами. Сведения, представленные здесь, относятся к операциям более низкого уровня.

Большинство модулей Python, предназначенных для организации взаимодействий с операционной системой, основаны на интерфейсах POSIX. POSIX – это стандарт, который определяет базовый набор интерфейсов операционных систем. Большинство версий UNIX поддерживают стандарт POSIX, а другие системы, такие как Windows, поддерживают значительную долю этих интерфейсов. На протяжении всей главы особо будут отмечаться функции и модули, которые могут использоваться только на определенной платформе. Под системами UNIX здесь также подразумеваются Linux и Mac OS X. Под Windows подразумеваются все версии Windows, если явно не оговаривается иное.

Читателям может потребоваться получить справочную информацию, дополняющую сведения, которые приводятся здесь. Отличный обзор файлов, дескрипторов файлов и низкоуровневых интерфейсов, на которых основаны многие модули, рассматриваемые в этом разделе, можно найти во втором издании книги «The C Programming Language» Брайана У. Кернигана (Brian W. Kernighan) и Денниса М. Ричи (Dennis M. Ritchie) (Prentice Hall, 1989).[1] Более опытным читателям можно порекомендовать второе издание

[1] Керниган Б., Ричи Д. «Язык программирования С», 2-е издание. – Пер. с англ. – Вильямс, 2008.

книги «Advanced Programming in the UNIX Environment» Ричарда У. Стивенса (W. Richard Stevens) и Стивена Раго (Stephen Rago) (Addison Wesley, 2005).[1] Краткий обзор общих понятий можно найти в учебнике по операционным системам. Однако, учитывая высокую стоимость и ограниченную практическую ценность этих книг, лучше будет, пожалуй, попросить их на выходные у знакомого студента, изучающего информационные технологии.

Модуль commands

Модуль `commands` применяется для выполнения простых системных команд, передавая их в виде строки и получая результаты их работы также в виде строки. Этот модуль может использоваться только в системах UNIX. Функциональные возможности, предоставляемые этим модулем, напоминают действие обратных апострофов (`) в сценариях командной оболочки UNIX. Например, инструкция `x = commands.getoutput('ls -l')` похожа на команду ``x=`ls -l` ``.

`getoutput(`*`cmd`*`)`

Выполняет команду *cmd* в командной оболочке и возвращает строку, содержащую потоки стандартного вывода и стандартного вывода сообщений об ошибках команды.

`getstatusoutput(`*`cmd`*`)`

Похожа на функцию `getoutput()` за исключением того, что возвращает кортеж из двух элементов `(status, output)`, где поле *status* содержит код завершения, возвращаемый функцией `os.wait()`, а поле *output* – строку, которую возвращает `getoutput()`.

Примечания

Этот модуль доступен только в Python 2. В Python 3 обе упомянутые функции находятся в модуле `subprocess`.

Хотя этот модуль и может использоваться для выполнения простых команд командной оболочки, тем не менее практически всегда лучше использовать модуль `subprocess`, позволяющий запускать дочерние процессы и получать их вывод.

См. также

Описание модуля `subprocess` (стр. 503).

Модули ConfigParser и configparser

Модуль `ConfigParser` (в Python 3 он называется `configparser`) используется для чтения файлов с настройками в формате файлов Windows INI. Эти

[1] Стивенс У., Раго С. «UNIX. Профессиональное программирование», 2-е издание. – Пер. с англ. – СПб.: Символ-Плюс, 2007.

файлы состоят из именованных разделов, в каждом из которых перечислены собственные переменные с их значениями, как показано ниже:

```
# Комментарий
; Комментарий
[section1]
name1 = value1
name2 = value2

[section2]
; Альтернативный синтаксис присваивания значений
name1: value1
name2: value2
...
```

Класс ConfigParser

Класс `ConfigParser` используется для работы с параметрами настройки:

`ConfigParser([defaults [, dict_type]])`

Создает новый экземпляр класса `ConfigParser`. В аргументе *defaults* можно передать необязательный словарь значений, на которые допускается ссылаться в параметрах настройки с помощью спецификаторов формата, таких как '%(*key*)s', где *key* – ключ в словаре *defaults*. Аргумент *dict_type* определяет тип словаря, который будет использоваться внутренней реализацией для хранения параметров настройки. По умолчанию используется тип `dict` (встроенный словарь).

Экземпляр *c* класса `ConfigParser` обладает следующими методами:

c.`add_section(`*section*`)`

Добавляет новый раздел для хранения параметров настройки. В аргументе *section* передается строка с именем раздела.

c.`defaults()`

Возвращает словарь со значениями по умолчанию.

c.`get(`*section*, *option* `[, raw [, vars]])`

Возвращает значение параметра *option* из раздела *section* в виде строки. По умолчанию возвращаемая строка проходит этап интерпретации, где выполняется подстановка фактического значения в строку формата, такую как '%(*option*)s'. В данном случае *option* может быть именем другого параметра настройки в том же разделе или одним из значений по умолчанию, переданных функции `ConfigParser` в аргументе *defaults*. В аргументе *raw* передается логический флаг, запрещающий интерпретацию, и в этом случае значение параметра *option* возвращается в первоначальном виде. В необязательном аргументе *vars* передается словарь с дополнительными значениями для использования оператором форматирования '%'.

`c.getboolean(`*section*, *option*`)`

Возвращает значение параметра *option* из раздела *section*, преобразованное в логическое значение. Метод распознает такие значения, как "0", "true",

"yes", "no", "on" и "off", без учета регистра символов. Этот метод всегда выполняет интерпретацию параметров (смотрите *c*.get()).

c.getfloat(*section*, *option*)

Возвращает значение параметра *option* из раздела *section*, преобразованное в число с плавающей точкой, с промежуточной интерпретацией значения параметра.

c.*getint*(*section*, *option*)

Возвращает значение параметра *option* из раздела *section*, преобразованное в целое число, с промежуточной интерпретацией значения параметра.

c.has_option(*section*, *option*)

Возвращает True, если раздел *section* содержит параметр с именем *option*.

c.has_section(*section*)

Возвращает True, если существует раздел с именем *section*.

c.items(*section* [, *raw* [, *vars*]])

Возвращает список пар (*option*, *value*) из раздела *section*. В аргументе *raw* передается логический флаг, запрещающий интерпретацию, если он имеет значение True. В аргументе *vars* передается словарь с дополнительными значениями для использования оператором форматирования '%'.

c.options(*section*)

Возвращает список всех параметров в разделе *section*.

c.optionxform(*option*)

Преобразует имя параметра *option* в строку, которая может использоваться для ссылки на параметр. По умолчанию приводит все символы к нижнему регистру.

c.read(*filenames*)

Читает параметры настройки из списка *filenames* и сохраняет их. Аргумент *filenames* может быть единственной строкой, и тогда он будет интерпретироваться как имя файла, который требуется прочитать, либо список имен файлов. Если какой-либо из указанных файлов не будет найден, он будет просто проигнорирован. Этот метод удобно использовать для чтения файлов с настройками из нескольких возможных местоположений, где такие файлы могут присутствовать или отсутствовать. Возвращает список имен успешно прочитанных файлов.

c.readfp(*fp* [, *filename*])

Читает параметры настройки из объекта *fp*, похожего на объект файла, который уже должен быть открыт. Аргумент *filename* определяет имя файла, ассоциированное с объектом *fp* (если необходимо). По умолчанию имя файла извлекается из атрибута *fp*.name или принимает значение '<???>', если этот атрибут не определен.

c.remove_option(*section*, *option*)

Удаляет параметр *option* из раздела *section*.

c.remove_section(*section*)

Удаляет раздел *section*.

c.sections()

Возвращает список имен всех разделов.

c.set(*section*, *option*, *value*)

Устанавливает параметр настройки *option*, из раздела *section*, в значение *value*. Значение *value* должно быть строкой.

c.write(*file*)

Записывает все параметры настройки, хранящиеся в текущий момент, в файл *file*. Аргумент *file* должен быть уже открытым объектом, похожим на объект файла.

Пример

Модуль ConfigParser часто упускается из виду, но это чрезвычайно полезный инструмент, позволяющий управлять программами со сложными пользовательскими настройками или окружением времени выполнения. Например, когда пишется компонент, который будет работать внутри более крупного приложения, файл с настройками зачастую является единственным элегантным способом передать ему параметры времени выполнения. Точно так же файл с параметрами настройки может оказаться более элегантным способом, чем передача огромного количества параметров в командной строке и чтение их с помощью модуля optparse. Существуют также тонкие, но значимые отличия между использованием файлов с настройками и чтением параметров из сценария на языке Python.

Ниже приводятся несколько примеров, иллюстрирующих некоторые из наиболее интересных особенностей модуля ConfigParser. Для начала взгляните на пример файла .ini:

```
# appconfig.ini
# Файл с параметрами настройки приложения mondo

[output]
LOGFILE=%(LOGDIR)s/app.log
LOGGING=on
LOGDIR=%(BASEDIR)s/logs

[input]
INFILE=%(INDIR)s/initial.dat
INDIR=%(BASEDIR)s/input
```

Следующий фрагмент программного кода демонстрирует, как читать содержимое файла с параметрами и назначать значения по умолчанию некоторым переменным:

```
from configparser import ConfigParser # Use from ConfigParser in Python 2

# Словарь со значениями по умолчанию
defaults = {
    'basedir' : '/Users/beazley/app'
```

```
}

# Создать объект ConfigParser и прочитать файл .ini
cfg = ConfigParser(defaults)
cfg.read('appconfig.ini')
```

После того как файл будет прочитан, значения параметров можно получать с помощью метода `get()`. Например:

```
>>> cfg.get('output','logfile')
'/Users/beazley/app/logs/app.log'
>>> cfg.get('input','infile')
'/Users/beazley/app/input/initial.dat'
>>> cfg.getboolean('output','logging')
True
>>>
```

Здесь сразу бросаются в глаза некоторые интересные особенности. Во-первых, имена параметров нечувствительны к регистру символов. То есть, если программа читает параметр с именем `'logfile'`, не имеет никакого значения, как он будет называться в файле с настройками, – `'logfile'`, `'LOGFILE'` или `'LogFile'`. Во-вторых, параметры настройки могут ссылаться на другие параметры, например `'%(BASEDIR)s'` и `'%(LOGDIR)s'`, присутствующие в файле. В данном случае имена параметров также нечувствительны к регистру символов. Кроме того, в случае использования ссылок на переменные, порядок их определения не имеет значения. Например, параметр `LOGFILE`, в файле `appconfig.ini`, ссылается на параметр `LOGDIR`, который определяется ниже. Наконец, значения в файлах параметров часто интерпретируются правильно, даже если они не совсем точно соответствуют синтаксису языка Python или типам данных. Например, значение `'on'` для параметра `LOGGING` интерпретируется методом `cfg.getboolean()`, как `True`.

Помимо этого, имеется возможность объединять файлы параметров. Например, допустим, что у пользователя имеется собственный файл с его настройками:

```
; userconfig.ini
;
; Настройки пользователя

[output]
logging=off

[input]
BASEDIR=/tmp
```

Имеется возможность объединить содержимое этого файла с уже загруженными параметрами настройки. Например:

```
>>> cfg.read('userconfig.ini')
['userconfig.ini']
>>> cfg.get('output','logfile')
'/Users/beazley/app/logs/app.log'
>>> cfg.get('output','logging')
'off'
```

```
>>> cfg.get('input','infile')
'/tmp/input/initial.dat'
>>>
```

Здесь видно, что вновь загруженные параметры выборочно заменили параметры, которые были загружены ранее. Кроме того, если изменить один из параметров настройки, который используется в других параметрах, действие этих изменений распространится и на другие параметры. Например, ввод нового значения параметра `BASEDIR` в разделе `input` отразится на параметрах настройки в этом разделе, которые были определены ранее, такие как `INFILE`. Такое поведение является важным, хотя и малозаметным отличием между использованием файлов с настройками и определением настроек программы в сценарии на языке Python.

Примечания

Вместо класса `ConfigParser` можно использовать два других класса. Класс `RawConfigParser` предоставляет аналогичные возможности, что и класс `ConfigParser`, но он не позволяет использовать ссылки на параметры. Класс `SafeConfigParser` также предоставляет аналогичные возможности, что и класс `ConfigParser`, но он позволяет решить проблему, когда значения параметров настройки включают в себя литералы специальных символов форматирования, используемые механизмом интерпретации (например, `'%'`).

Модуль datetime

Модуль `datetime` объявляет различные классы для представления и работы с датой и временем. Значительные части этого модуля просто связаны с различными способами создания и манипулирования информацией о дате и времени. В число других важных особенностей входят математические операции, такие как сравнивание и вычисление разницы во времени. Работа с датами является сложным аспектом программирования, поэтому я настоятельно рекомендую читателям ознакомиться с электронной документацией Python, чтобы получить представление об организации этого модуля.

Объекты класса date

Объект класса `date` используется для представления простых дат и содержит атрибуты `year`, `month` и `day`. Следующие четыре функции позволяют создавать объекты типа `date`:

`date(`*`year`*`, `*`month`*`, `*`day`*`)`

Создает новый объект типа `date`. В аргументе *`year`* передается целое число в диапазоне от `datetime.MINYEAR` до `datetime.MAXYEAR`. В аргументе *`month`* передается целое число в диапазоне от 1 до 12, а в аргументе *`day`* – целое число в диапазоне от 1 до последнего числа указанного месяца *`month`*. Возвращает неизменяемый объект `date`, обладающий атрибутами `year`, `month` и `day`, в которых сохраняются значения соответствующих аргументов.

`date.today()`

Метод класса, который возвращает объект `date`, соответствующий текущей дате.

`date.fromtimestamp(`*`timestamp`*`)`

Метод класса, который возвращает объект `date`, соответствующий отметке времени *`timestamp`*. В аргументе *`timestamp`* передается значение, возвращаемое функцией `time.time()`.

`date.fromordinal(`*`ordinal`*`)`

Метод класса, который возвращает объект `date`, соответствующий дате, отстоящей на *`ordinal`* дней от минимально допустимой даты (дата 1 января 1 года имеет порядковый номер 1, а 1 января 2006 года имеет порядковый номер 732312).

Следующие атрибуты класса описывают минимальное и максимальное значения дат, а также разрешающую способность экземпляров класса `date`.

`date.min`

Атрибут класса, представляющий минимально допустимую дату (`datetime.date(1,1,1)`).

`date.max`

Атрибут класса, представляющий максимально допустимую дату (`datetime.date(9999,12,31)`).

`date.resolution`

Наименьшая различимая разница между двумя неравными объектами `date` (`datetime.timedelta(1)`).

Экземпляр *`d`* класса `date` обладает атрибутами *`d`*`.year`, *`d`*`.month` и *`d`*`.day`, доступными только для чтения, и дополнительно предоставляет следующие методы:

`d``.ctime()`

Возвращает строковое представление даты в том же формате, который обычно используется функцией `time.ctime()`.

`d``.isocalendar()`

Возвращает дату в виде кортежа (*`iso_year`*, *`iso_week`*, *`iso_weekday`*), где поле *`iso_week`* содержит число в диапазоне от 1 до 53, а поле *`iso_weekday`* – число в диапазоне от 1 (понедельник) до 7 (воскресенье). Значение 1 в поле *`iso_week`* соответствует первой неделе года, включающей четверг. Диапазон значений компонентов кортежа определяется стандартом ISO 8601.

`d``.isoformat()`

Возвращает строку вида `'YYYY-MM-DD'`, отформатированную в соответствии с требованиями стандарта ISO 8601, представляющую дату.

`d``.isoweekday()`

Возвращает день недели в диапазоне от 1 (понедельник) до 7 (воскресенье).

d.replace([*year* [, *month* [, *day*]]])

Возвращает новый объект класса date, в котором один или более атрибутов получают новые значения. Например, вызов *d*.replace(month=4) вернет новый объект date, в котором атрибут month получит новое значение 4.

d.strftime(*format*)

Возвращает строковое представление даты, отформатированное в соответствии с правилами, которые применяются при работе с функцией time.strftime(). Эта функция может работать только с датами позднее 1900 года. Кроме того, спецификаторы формата для компонентов, отсутствующих в объектах типа date (таких как часы, минуты и так далее), не должны использоваться.

d.timetuple()

Возвращает объект time.struct_time, пригодный для использования в вызовах функций из модуля time. Значения, имеющие отношение к времени суток (часы, минуты, секунды), будут установлены в 0.

d.toordinal()

Возвращает порядковый номер даты *d*. Дата 1 января 1 года имеет порядковый номер 1.

d.weekday()

Возвращает день недели в диапазоне от 0 (понедельник) до 6 (воскресенье).

Объекты класса time

Объекты класса time используются для представления времени в часах, минутах, секундах и микросекундах. Создание объектов производится с помощью следующего конструктора класса:

time(*hour* [, *minute* [, *second* [, *microsecond* [, *tzinfo*]]]])

Создает объект класса time, представляющий время суток, где для значений аргументов должны соблюдаться следующие условия: 0 <= *hour* < 24, 0 <= *minute* < 60, 0 <= *second* < 60 и 0 <= *microsecond* < 1000000. Аргумент *tzinfo* является экземпляром класса tzinfo, который описывается ниже, в этом же разделе, и используется для передачи информации о часовом поясе. Возвращаемый объект time обладает атрибутами hour, minute, second, microsecond и tzinfo, в которых сохраняются значения соответствующих аргументов.

Следующие атрибуты класса time описывают диапазоны допустимых значений и разрешающую способность экземпляров класса time:

time.min

Атрибут класса, представляющий минимально допустимое время суток (datetime.time(0,0)).

time.max

Атрибут класса, представляющий максимально допустимое время суток (datetime.time(23,59,59, 999999)).

`time.resolution`

Наименьшая различимая разница между двумя неравными объектами time (datetime.timedelta(0,0,1)).

Экземпляр *t* класса time, помимо атрибутов *t*.hour, *t*.minute, *t*.second, *t*.microsecond и *t*.tzinfo, обладает также следующими методами:

t.dst()

Возвращает значение *t*.tzinfo.dst(None). Возвращаемое значение является объектом класса timedelta. Если информация о часовом поясе отсутствует, возвращает None.

t.isoformat()

Возвращает строковое представление времени вида 'HH:MM:SS.mmmmmm'. Если число микросекунд равно нулю, часть строки '.mmmmmm' опускается. Если в экземпляре присутствует информация о часовом поясе, к представлению времени может быть добавлено смещение часового пояса (например, 'HH:MM:SS.mmmmmm+HH:MM').

t.replace([*hour* [, *minute* [, *second* [, *microsecond* [, *tzinfo*]]]]])

Возвращает новый объект класса time, в котором один или более атрибутов получают новые значения. Например, вызов *t*.replace(second=30) вернет новый объект time, в котором атрибут seconds получит новое значение 30. Аргументы имеют тот же смысл, что и в функции time(), описанной выше.

t.strftime(*format*)

Возвращает строковое представление времени, отформатированное в соответствии с правилами, которые применяются при работе с функцией time.strftime() из модуля time. Так как объекты time не содержат информацию о дате, то допускается использовать только спецификаторы формата, имеющие отношение к времени суток.

t.tzname()

Возвращает значение t.tzinfo.tzname(). Если информация о часовом поясе отсутствует, возвращает None.

t.utcoffset()

Возвращает значение t.tzinfo.utcoffset(None). Возвращаемое значение является объектом класса timedelta. Если информация о часовом поясе отсутствует, возвращает None.

Объекты класса datetime

Объекты класса datetime используются для представления даты и времени. Существует множество способов создания экземпляров класса datetime:

datetime(*year*, *month*, *day* [, *hour* [, *minute* [, *second* [, *microsecond* [, *tzinfo*]]]]])

Создает объект класса datetime, обладающий всеми возможностями объектов классов date и time. Аргументы имеют тот же смысл, что и в функциях date() и time().

`datetime.combine(`*`date`*`,`*`time`*`)`

Метод класса, который создает объект `datetime`, объединяя содержимое объекта `date`, представленного аргументом *`date`*, и объекта `time`, представленного аргументом *`time`*.

`datetime.fromordinal(`*`ordinal`*`)`

Метод класса, который создает объект `datetime`, соответствующий дате, отстоящей на *`ordinal`* дней от минимально допустимой даты (целое число дней, прошедших с даты `datetime.min`). Все атрибуты, определяющие время суток, устанавливаются в значение 0, а атрибут `tzinfo` – в значение `None`.

`datetime.fromtimestamp(`*`timestamp`* `[,` *`tz`*`])`

Метод класса, который создает объект `datetime`, соответствующий отметке времени *`timestamp`*, возвращаемой функцией `time.time()`. В необязательном аргументе *`tz`*, который является экземпляром класса `tzinfo`, передается информация о часовом поясе.

`datetime.now([`*`tz`*`])`

Метод класса, который создает объект `datetime`, соответствующий текущей локальной дате и времени. В необязательном аргументе *`tz`*, который является экземпляром класса `tzinfo`, передается информация о часовом поясе.

`datetime.strptime(`*`datestring`*`,` *`format`*`)`

Метод класса, который создает объект `datetime` из строки *`datestring`*, руководствуясь форматом в аргументе *`format`*. Анализ строки выполняется с помощью функции `strptime()` из модуля `time`.

`datetime.utcfromtimestamp(`*`timestamp`*`)`

Метод класса, который создает объект `datetime`, соответствующий отметке времени *`timestamp`*, обычно возвращаемой функцией `time.gmtime()`.

`datetime.utcnow()`

Метод класса, который создает объект `datetime`, соответствующий текущей дате и времени по Гринвичу.

Следующие атрибуты класса описывают минимальное и максимальное значения дат и времени, а также разрешающую способность экземпляров класса `datetime`.

`datetime.min`

Атрибут класса, представляющий минимально допустимую дату и время (`datetime.datetime(1,1,1,0,0)`).

`datetime.max`

Атрибут класса, представляющий максимально допустимую дату и время (`datetime.datetime(9999,12,31,23,59,59,999999)`).

`datetime.resolution`

Наименьшая различимая разница между двумя неравными объектами (`datetime.timedelta(0,0,1)`).

Экземпляр *d* класса datetime обладает теми же методами, что и экземпляры классов date и time вместе взятые. Кроме того, он имеет следующие методы:

d.astimezone(*tz*)

Возвращает новый объект datetime, но с другим часовым поясом *tz*. Атрибуты нового объекта будут соответствовать тому же времени по Гринвичу, но в другом часовом поясе *tz*.

d.date()

Возвращает объект date с той же датой.

d.replace([*year* [, *month* [, *day* [, *hour* [, *minute* [, *second* [, *microsecond* [, *tzinfo*]]]]]]])

Возвращает новый объект datetime, в котором значения одного или более атрибутов установлены в соответствии со значениями аргументов. Для замены отдельных значений можно использовать именованные аргументы.

d.time()

Возвращает объект time с тем же временем. Возвращаемый объект не содержит информацию о часовом поясе.

d.timetz()

Возвращает объект time с тем же временем и с тем же часовым поясом.

d.utctimetuple()

Возвращает объект time.struct_time, содержащий дату и время, приведенные к Гринвичскому времени.

Объекты класса timedelta

Объекты класса timedelta представляют разницу между двумя датами или значениями времени. Эти объекты обычно создаются при вычислении разности между двумя экземплярами класса datetime с помощью оператора вычитания (-). Однако имеется возможность создавать их и вручную, с помощью следующего конструктора класса:

timedelta([*days* [, *seconds* [, *microseconds* [, *milliseconds* [, *minutes* [, *hours* [, *weeks*]]]]]]])

Создает объект timedelta, представляющий разницу между двумя датами и значениями времени. Значимыми являются только аргументы *days*, *seconds* и *microseconds*, которые используются внутренней реализацией для представления разности. Остальные аргументы, если они указаны, преобразуются в дни, секунды и микросекунды. Эти значения сохраняются в атрибутах days, seconds и microseconds возвращаемого объекта timedelta.

Следующие атрибуты класса описывают минимальное и максимальное значение, а также разрешающую способность экземпляров класса timedelta:

timedelta.min

Самая большая отрицательная разность, которая может быть представлена объектом timedelta (timedelta(-999999999)).

`timedelta.max`

Самая **большая положительная разность, которая может быть представлена объектом** `timedelta` **(**`timedelta(days=999999999, hours=23, minutes=59, seconds=59, microseconds=999999)`**).**

`timedelta.resolution`

Наименьшая различимая разница между двумя неравными объектами `timedelta` **(**`timedelta(microseconds=1)`**).**

Математические операции над датами

Важной особенностью модуля `datetime` является поддержка математических операций над датами. Оба класса, `date` и `datetime`, поддерживают следующие операции:

Операция	Описание
`td = date1 - date2`	Возвращает объект `timedelta`
`date2 = date1 + td`	Добавляет разность `timedelta` к объекту `date`
`date2 = date1 - td`	Вычитает разность `timedelta` из объекта `date`
`date1 < date2`	Сравнивание дат
`date1 <= date2`	
`date1 == date2`	
`date1 != date2`	
`date1 > date2`	
`date1 >= date2`	

Когда возникает необходимость сравнить две даты, не следует забывать о часовом поясе. Если объект с датой содержит информацию о часовом поясе, его можно будет сравнивать с другим объектом даты, только если он также содержит информацию о часовом поясе; в противном случае будет возбуждено исключение `TypeError`. Когда выполняется сравнивание двух дат в разных часовых поясах, они сначала приводятся к гринвичскому времени, и только потом сравниваются. Объекты `timedelta` также поддерживают различные математические операции:

Операция	Описание
`td3 = td2 + td1`	Складывает две разности
`td3 = td2 - td1`	Вычитает одну разность из другой
`td2 = td1 * i`	Умножает на целое число
`td2 = i * td1`	
`td2 = td1 // i`	Целочисленное деление на `i` с округлением вниз
`td2 = -td1`	Унарные минус и плюс

(продолжение)

Операция	Описание
td2 = +td1	
abs(td)	Абсолютное значение
td1 < td2	Сравнивание
td1 <= td2	
td1 == td2	
td1 != td2	
td1 > td2	
td1 >= td2	

Ниже приводится несколько примеров:

```
>>> today = datetime.datetime.now()
>>> today.ctime()
'Thu Oct 20 11:10:10 2005'
>>> oneday = datetime.timedelta(days=1)
>>> tomorrow = today + oneday
>>> tomorrow.ctime()
'Fri Oct 21 11:10:10 2005'
>>>
```

Кроме того, все объекты классов `date`, `datetime`, `time` и `timedelta` являются неизменяемыми. Это означает, что они могут использоваться в качестве ключей словаря, добавляться в множества и использоваться в других операциях.

Объекты класса tzinfo

Многие из методов в модуле `datetime` выполняют операции над объектами специального класса `tzinfo`, который используется для представления информации о часовом поясе. Класс `tzinfo` – это просто базовый класс. Отдельные часовые пояса созданы как классы, производные от `tzinfo`, и реализуют следующие методы:

tz.dst(*dt*)

Возвращает объект `timedelta`, представляющий разность для перехода на летнее время, если это возможно. Если информация о летнем времени недоступна, возвращает `None`. В аргументе *dt* должен передаваться либо объект `datetime`, либо `None`.

tz.fromutc(*dt*)

Преобразует объект *dt* класса `datetime` из гринвичского времени в локальное и возвращает новый объект `datetime`. Этот метод вызывается методом `astimezone()` объектов `datetime`. Реализация по умолчанию этого метода уже имеется в классе `tzinfo`, поэтому обычно нет необходимости переопределять этот метод.

tz.tzname(*dt*)

Возвращает строку с названием часового пояса (например, "US/Central"). В аргументе *dt* должен передаваться либо объект datetime, либо None.

tz.utcoffset(*dt*)

Возвращает объект timedelta, представляющий смещение в минутах локального времени на восток от Гринвича. Смещение включает в себя все элементы, составляющие локальное время, включая разницу между зимним и летним временем, если это возможно. В аргументе *dt* должен передаваться либо объект datetime, либо None.

Следующий пример демонстрирует основные принципы объявления часового пояса:

```
# Переменные, которые должны быть определены
# TZOFFSET - смещение в часах часового пояса относительно Гринвича.
# Например, смещение для часового пояса US/CST составляет -6 часов
# DSTNAME - название часового пояса,
#           предусматривающего переход на летнее время
# STDNAME - название часового пояса,
#           не предусматривающего переход на летнее время

class SomeZone(datetime.tzinfo):
    def utcoffset(self,dt):
        return datetime.timedelta(hours=TZOFFSET) + self.dst(dt)
    def dst(self,dt):
        # is_dst() - это функция, которую вы должны реализовать сами,
        # чтобы иметь возможность проверить, предусматривается ли
        # данным часовым поясом переход на летнее время.
        if is_dst(dt):
            return datetime.timedelta(hours=1)
        else:
            return datetime.timedelta(0)
    def tzname(self,dt):
        if is_dst(dt):
            return DSTNAME
        else:
            return STDNAME
```

Множество примеров объявления часовых поясов можно также найти в электронной документации с описанием модуля datetime.

Анализ строк с датой и временем

При работе с датой и временем часто возникает проблема преобразования различных строковых представлений даты и времени в соответствующие объекты datetime. В модуле datetime имеется единственная функция datetime.strptime(), которая обрабатывает такие строковые представления. Однако, чтобы воспользоваться ею, необходимо точно определить формат представления даты в строке, используя различные спецификаторы формата (смотрите описание функции time.strptime()). Например, чтобы преобразовать строку s="Aug 23, 2008", пришлось бы написать вызов d = datetime.datetime.strptime(s, "%b %d, %Y").

Если вам потребуется интеллектуальный механизм преобразования дат, который автоматически распознает наиболее типичные форматы, придется обратиться к сторонним модулям. Посетите сайт каталога пакетов Python (*http://pypi.python.org*) и выполните поиск по слову «datetime». Вам будет представлено широкое разнообразие вспомогательных модулей, расширяющих возможности модуля `datetime`.

См. также

Описание модуля `time` (стр. 507).

Модуль errno

Модуль `errno` определяет символические имена для целочисленных кодов ошибок, возвращаемых различными системными вызовами, в частности теми, которые можно найти в модулях `os` и `socket`. Обычно эти коды встречаются в атрибуте `errno` исключения `OSError` или `IOError`. Для преобразования кода ошибки в строку с текстом сообщения можно использовать функцию `os.strerror()`. А для преобразования целочисленных кодов ошибок в символические имена можно использовать следующий словарь:

`errorcode`

Этот словарь отображает целочисленные коды ошибок в символические имена (такие как `'EPERM'`).

Коды ошибок в стандарте POSIX

В следующей таблице перечислены символические имена, определяемые стандартом POSIX для обозначения типичных кодов системных ошибок. Коды ошибок, перечисленные здесь, поддерживаются большинством версий UNIX, Macintosh OS-X и Windows. Некоторые версии UNIX могут определять дополнительные коды ошибок. Они встречаются достаточно редко и потому не были включены в список. Если возникает подобная ошибка, можно обратиться к словарю `errorcode`, определить с его помощью соответствующее символическое имя и использовать его в своей программе.

Код ошибки	Описание
`E2BIG`	Слишком длинный список аргументов.
`EACCES`	В разрешении отказано.
`EADDRINUSE`	Адрес уже используется.
`EADDRNOTAVAIL`	Невозможно назначить запрошенный адрес.
`EAFNOSUPPORT`	Семейство адресов не поддерживается протоколом.
`EAGAIN`	Требуется повторить попытку.
`EALREADY`	Операция уже выполняется.
`EBADF`	Неверный номер файла.

Код ошибки	Описание
EBUSY	Устройство или ресурс занят.
ECHILD	Нет дочернего процесса.
ECONNABORTED	Соединение было разорвано.
ECONNREFUSED	Соединение было отвергнуто.
ECONNRESET	Соединение было сброшено удаленным узлом.
EDEADLK	Возникла ситуация взаимоблокировки при попытке захватить ресурс.
EDEADLOCK	Возникла ситуация взаимоблокировки при попытке захватить файл.
EDESTADDRREQ	Не указан адрес назначения.
EDOM	Математический аргумент за пределами области определения функции.
EDQUOT	Превышена квота.
EEXIST	Файл существует.
EFAULT	Недопустимый адрес.
EFBIG	Файл слишком большой.
EHOSTDOWN	Хост не работает.
EHOSTUNREACH	Нет доступных маршрутов к хосту.
EILSEQ	Недопустимая последовательность байтов.
EINPROGRESS	Операция теперь выполняется.
EINTR	Системный вызов был прерван.
EINVAL	Недопустимый аргумент.
EIO	Ошибка ввода-вывода.
EISCONN	Сокет уже подключен.
EISDIR	Это каталог.
ELOOP	Встречено слишком много символических ссылок.
EMFILE	Слишком много открытых файлов.
EMLINK	Слишком много ссылок.
EMSGSIZE	Сообщение слишком длинное.
ENETDOWN	Сеть не работает.
ENETRESET	Отказ в соединении от сети.
ENETUNREACH	Сеть недоступна.
ENFILE	Таблица файлов переполнена.
ENOBUFS	Недостаточно места в буфере.
ENODEV	Нет такого устройства.

(продолжение)

Код ошибки	Описание
ENOENT	Нет такого файла или каталога.
ENOEXEC	Ошибка формата выполняемого файла.
ENOLCK	Нет доступных блокировок.
ENOMEM	Недостаточно памяти.
ENOPROTOOPT	Протокол недоступен.
ENOSPC	Недостаточно места на устройстве.
ENOSYS	Функция не реализована.
ENOTCONN	Сокет не подключен.
ENOTDIR	Не каталог.
ENOTEMPTY	Непустой каталог.
ENOTSOCK	Не является сокетом.
ENOTTY	Не является терминалом.
ENXIO	Неправильный адрес или устройство.
EOPNOTSUPP	Операция не поддерживается для сокетов.
EPERM	Операция запрещена.
EPFNOSUPPORT	Семейство протоколов не поддерживается.
EPIPE	Разрушенный канал.
EPROTONOSUPPORT	Протокол не поддерживается.
EPROTOTYPE	Недопустимый тип протокола для сокета.
ERANGE	Результат математической операции слишком велик.
EREMOTE	Попытка обращения к удаленному объекту.
EROFS	Файловая система доступна только для чтения.
ESHUTDOWN	Невозможно выполнить передачу после отключения сокета.
ESOCKTNOSUPPORT	Тип сокета не поддерживается.
ESPIPE	Неверное позиционирование.
ESRCH	Нет такого процесса.
ESTALE	Устаревший дескриптор файла в файловой системе NFS.
ETIMEDOUT	Превышено время ожидания соединения.
ETOOMANYREFS	Слишком много ссылок, невозможно отследить.
EUSERS	Слишком много пользователей.
EWOULDBLOCK	Операция может быть заблокирована.
EXDEV	Ссылка между устройствами.

Коды ошибок в Windows

Коды ошибок, перечисленные в следующей таблице, доступны только в Windows.

Код ошибки	Описание
WSAEACCES	В разрешении отказано.
WSAEADDRINUSE	Адрес уже используется.
WSAEADDRNOTAVAIL	Невозможно назначить запрошенный адрес.
WSAEAFNOSUPPORT	Семейство адресов не поддерживается протоколом.
WSAEALREADY	Операция уже выполняется.
WSAEBADF	Неверный дескриптор файла.
WSAECONNABORTED	Соединение было разорвано.
WSAECONNREFUSED	Соединение было отвергнуто.
WSAECONNRESET	Соединение было сброшено удаленным узлом.
WSAEDESTADDRREQ	Не указан адрес назначения.
WSAEDISCON	Удаленный хост не работает.
WSAEDQUOT	Превышена квота на дисковое пространство.
WSAEFAULT	Недопустимый адрес.
WSAEHOSTDOWN	Хост не работает.
WSAEHOSTUNREACH	Нет доступных маршрутов к хосту.
WSAEINPROGRESS	Операция в процессе выполнения.
WSAEINTR	Системный вызов был прерван.
WSAEINVAL	Недопустимый аргумент.
WSAEISCONN	Сокет уже подключен.
WSAELOOP	Невозможно преобразовать имя.
WSAEMSGSIZE	Сообщение слишком длинное.
WSAENAMETOOLONG	Слишком длинное имя.
WSAENETDOWN	Сеть не работает.
WSAENETRESET	Отказ в соединении от сети.
WSAENETUNREACH	Сеть недоступна.
WSAENOBUFS	Недостаточно места в буфере.
WSAENOPROTOOPT	Недопустимый параметр протокола.
WSAENOTCONN	Сокет не подключен.
WSAENOTEMPTY	Невозможно удалить непустой каталог.
WSAENOTSOCK	Не является сокетом.

(продолжение)

Код ошибки	Описание
WSAEOPNOTSUPP	Операция не поддерживается.
WSAEPFNOSUPPORT	Семейство протоколов не поддерживается.
WSAEPROCLIM	Слишком много процессов.
WSAEPROTONOSUPPORT	Протокол не поддерживается.
WSAEPROTOTYPE	Недопустимый тип протокола для сокета.
WSAEREMOTE	Объект не является локальным.
WSAESHUTDOWN	Невозможно выполнить передачу после отключения сокета.
WSAESOCKTNOSUPPORT	Тип сокета не поддерживается.
WSAESTALE	Дескриптор файла больше недоступен.
WSAETIMEDOUT	Превышено время ожидания соединения.
WSAETOOMANYREFS	Слишком много ссылок на объект ядра.
WSAEUSERS	Слишком много пользователей.
WSAEWOULDBLOCK	Ресурс временно не доступен.
WSANOTINITIALISED	Интерфейс сокетов не был инициализирован.
WSASYSNOTREADY	Сетевая подсистема недоступна.
WSAVERNOTSUPPORTED	Версия Winsock.dll не поддерживается.

Модуль fcntl

Модуль `fcntl` реализует управление файлами и операциями ввода-вывода на уровне дескрипторов файлов в системе UNIX. Дескриптор файла можно получить с помощью метода `fileno()` объекта файла или сокета.

`fcntl(`*`fd`*`, `*`cmd`*` [, `*`arg`*`])`

Выполняет команду *cmd* над дескриптором открытого файла *fd*. В аргументе *cmd* передается целочисленный код команды. В необязательном аргументе *arg* можно передать дополнительный параметр команды *cmd*, который может быть целым числом или строкой. Если в аргументе *arg* передается целое число, возвращаемым значением функции также будет целое число. Если в аргументе *arg* передается строка, она будет интерпретироваться как структура двоичных данных, а возвращаемое значение будет представлять собой содержимое буфера, преобразованное в объект строки. В этом случае длина аргумента *arg* и длина возвращаемого значения не должны превышать 1024 байтов, чтобы избежать возможного повреждения данных. Ниже приводится перечень доступных команд:

Команда	Описание
F_DUPFD	Создает дубликат дескриптора файла. В аргументе *arg* передается минимальное число, которое может быть назначено новому дескриптору файла. По своему действию эта команда аналогична системному вызову os.dup().
F_SETFD	Устанавливает флаг close-on-exec в значение, переданное в аргументе *arg* (0 или 1). Если установить этот флаг, файл будет закрыт системным вызовом exec().
F_GETFD	Возвращает состояние флага close-on-exec.
F_SETFL	Устанавливает флаг состояния в значение, переданное в аргументе *arg*, которое является битовой маской, составленной из следующих флагов, объединяемых битовой операцией ИЛИ: O_NDELAY – Неблокирующий режим ввода-вывода (System V) O_APPEND – Режим добавления в конец файла (System V) O_SYNC – Синхронный режим записи (System V) FNDELAY – Неблокирующий режим ввода-вывода (BSD) FAPPEND – Режим добавления в конец файла (BSD) FASYNC – Передает группе процессов сигнал SIGIO, когда операции ввода-вывода станут возможными (BSD).
F_GETFL	Возвращает флаг состояния, установленный командой F_SETFL.
F_GETOWN	Возвращает идентификатор процесса или группы процессов, в настоящее время получающих сигналы SIGURG и SIGIO (BSD).
F_SETOWN	Назначает идентификатор процесса или группы процессов, которые будут получать сигналы SIGIO и SIGURG (BSD).
F_GETLK	Возвращает структуру flock, используемую в операции блокировок файлов.
F_SETLK	Устанавливает блокировку на файл, возвращает -1, если блокировка уже была установлена в другом месте программы или другим процессом.
F_SETLKW	Устанавливает блокировку на файл, и если файл уже был где-то заблокирован, ожидает, пока прежняя блокировка не будет снята.

Если функция fcntl() терпит неудачу, возбуждается исключение IOError. Поддержка команд F_GETLK и F_SETLK реализована на основе функции lockf().

ioctl(*fd*, *op*, *arg* [, *mutate_flag*])

Эта функция похожа на функцию fcntl() за исключением того, что коды операций, которые передаются в аргументе *op*, определены в библиотечном модуле termios. Дополнительный флаг *mutate_flag* управляет поведением этой функции, когда ей в аргументе *arg* передается изменяемый объект буфера. Дополнительную информацию об этом можно найти в электронной документации. В основном функция ioctl() используется для организации взаимодействий с драйверами устройств и низкоуровневыми компонен-

тами, поэтому особенности ее применения тесно связаны с операционной системой. Она не должна использоваться в программах, для которых переносимость имеет большое значение.

`flock(`*`fd`*`, `*`op`*`)`

Выполняет операцию *op* блокировки над дескриптором *fd* файла. В аргументе *op* передается битная маска, составленная из значений следующих констант, объявленных в модуле `fcntl`, объединенных битовой операцией ИЛИ:

Флаг	Описание
`LOCK_EX`	Исключительный режим работы. Все последующие попытки установить блокировку будут блокироваться, пока блокировка не будет снята.
`LOCK_NB`	Неблокирующий режим работы. Возвращает управление немедленно, возбуждая исключение `IOError`, если блокировка уже была установлена.
`LOCK_SH`	Совместный режим работы. Блокирует любые попытки установить исключительную блокировку (`LOCK_EX`), но не запрещает установку блокировок совместного режима.
`LOCK_UN`	Снять блокировку. Снимает ранее установленную блокировку.

При попытке установить неблокирующий режим возбуждается исключение `IOError`, если блокировка не может быть установлена немедленно. В некоторых системах имеется возможность открыть файл и установить блокировку в одной операции, добавив специальный флаг в вызов функции `os.open()`. Дополнительные подробности можно найти в описании модуля `os`.

`lockf(`*`fd`*`, `*`op`*` [, `*`len`*` [, `*`start`*` [, `*`whence`*`]]])`

Устанавливает блокировку на фрагмент файла. В аргументе *op* передается то же самое значение, что и при вызове функции `flock()`. В аргументе *len* передается количество байтов, которые должны блокироваться. В аргументе *start* передается начальная позиция блокируемого фрагмента относительно значения в аргументе *whence*. Значение 0 в аргументе *whence* соответствует началу файла, 1 – текущей позиции и 2 – концу файла.

Пример

```
import fcntl

# Открыть файл
f = open("foo","w")

# Установить бит close-on-exec для объекта f файла
fcntl.fcntl(f.fileno(), fcntl.F_SETFD, 1)

# Установить исключительную блокировку на файл
fcntl.flock(f.fileno(), fcntl.LOCK_EX)
```

```
# Заблокировать первые 8192 байтов в файле (неблокирующий режим)
try:
    fcntl.lockf(f.fileno(), fcntl.LOCK_EX | fcntl.LOCK_NB, 8192, 0, 0)
except IOError,e:
    print "Невозможно установить блокировку", e
```

Примечания

- Множество команд и их параметров, доступных функции `fcntl()`, зависит от операционной системы. Для некоторых платформ модуль `fcntl` объявляет более 100 констант.
- В других модулях часто можно встретить операции блокировки, реализованные на основе протокола менеджеров контекста. Это далеко не то же самое, что и блокировка файлов. Если программа устанавливает блокировку на файл, она обязательно должна снимать ее.
- Многие функции из этого модуля могут также применяться к файловым дескрипторам сокетов.

Модуль io

Модуль `io` объявляет классы для различных операций ввода-вывода, а также встроенную функцию `open()`, используемую в Python 3. Этот модуль также доступен для использования в Python 2.6.

Основное назначение модуля `io` состоит в том, чтобы обеспечить единообразие реализации различных форм ввода-вывода. Например, работа с текстом несколько отличается от работы с двоичными данными, что обусловлено проблемами, связанными с символами перевода строки и кодировкой символов. Для устранения этих различий модуль организован как последовательность уровней, каждый из которых расширяет функциональные возможности предыдущего.

Базовый интерфейс ввода-вывода

Модуль `io` определяет базовый программный интерфейс операций ввода-вывода, который реализуют все объекты, похожие на файлы. Этот **интерфейс** объявлен как базовый класс `IOBase`. Экземпляр *f* класса `IOBase` поддерживает все основные операции ввода-вывода:

Атрибут	Описание
f.closed	Флаг, указывающий, был ли закрыт файл.
f.close()	Закрывает файл.
f.fileno()	Возвращает целое число дескриптора файла.
f.flush()	Выталкивает буферы ввода-вывода (если имеются).
f.isatty()	Возвращает `True`, если *f* представляет терминал.
f.readable()	Возвращает `True`, если *f* был открыт для чтения.

(продолжение)

Атрибут	Описание
f.readline([*limit*])	Читает одну строку из потока. Аргумент *limit* определяет максимальное число байтов для чтения.
f.readlines([*limit*])	Читает все строки из объекта *f* и возвращает их в виде списка. Аргумент *limit*, если он указан, определяет максимальное число байтов, которое может быть прочитано. Фактическое число прочитанных байтов может оказаться немного больше указанного, так как метод постарается прочитать последнюю уместившуюся строку до конца, чтобы не повредить ее.
f.seek(*offset*, [*whence*])	Перемещает указатель текущей позиции в новое место относительно местоположения, определяемого аргументом *whence*. В аргументе *offset* передается число байтов смещения. Значение 0 в аргументе *whence* соответствует началу файла, 1 – текущей позиции и 2 – концу файла.
f.seekable()	Возвращает True, если объект *f* поддерживает возможность произвольного перемещения указателя текущей позиции.
f.tell()	Возвращает текущую позицию указателя.
f.truncate([*size*])	Усекает файл до размера *size* байтов. Если аргумент *size* не указан, размер файла усекается до 0.
f.writable()	Возвращает True, если *f* был открыт для записи.
f.writelines(*lines*)	Записывает последовательность строк в *f*. Символы завершения строк не добавляются, поэтому они уже должны присутствовать в строках.

Низкоуровневый ввод–вывод

На самом низком уровне системы ввода-вывода находятся операции непосредственного ввода-вывода байтов. Основой этого уровня является класс FileIO, который обеспечивает достаточно прямолинейный интерфейс к низкоуровневым системным вызовам, таким как read() и write().

FileIO(*name* [, *mode* [, *closefd*]])

Возвращает объект, реализующий низкоуровневые операции ввода-вывода над файлом или дескриптором файла. В аргументе *name* передается имя файла или дескриптор файла, аналогичный тому, что возвращается функцией os.open() или методом fileno() объектов файлов. В аргументе *mode* передается одно из значений: 'r' (по умолчанию); 'w' или 'a', соответствующие режимам открытия файла для чтения, для записи или для добавления в конец. В аргумент *mode* можно добавить символ '+', который определяет режим открытия для изменения, когда поддерживаются обе операции – чтения и записи. В аргументе *closefd* передается флаг, который определяет, должен ли метод close() действительно закрывать сам файл. По умол-

чанию получает значение True, но его можно установить в значение False, если объект класса FileIO используется как обертка вокруг файла, открытого в другом месте программы. Если в аргументе *name* было передано имя файла, объект файла открывается с помощью системного вызова open(). Этот объект не поддерживает буферизацию данных и интерпретирует их как простые строки байтов. Экземпляр *f* класса FileIO может выполнять все основные операции ввода-вывода, описанные выше, и дополнительно обладает следующими атрибутами и методами:

Атрибут	Описание
f.closefd	Флаг, указывающий – будет ли закрыт дескриптор файла методом *f*.close() (только для чтения).
f.mode	Режим, выбранный при открытии файла (только для чтения).
f.name	Имя файла (только для чтения).
f.read([*size*])	Читает не более *size* байтов с помощью единственного обращения к системному вызову. Если аргумент *size* не указан, возвращает максимально возможное число байтов, прочитанных методом *f*.readall(). Эта операция может возвращать меньшее число байтов, чем указано в аргументе *size*, поэтому для проверки всегда следует использовать функцию len(). В неблокирующем режиме возвращает None, если данные недоступны.
f.readall()	Читает максимально возможное число байтов и возвращает их в виде одной строки. По достижении конца файла возвращает пустую строку. В неблокирующем режиме возвращает объем данных, которые доступны на данный момент.
f.write(*bytes*)	Записывает строку байтов или массив байтов *bytes* в *f*, с помощью единственного обращения к системному вызову. Возвращает число байтов, которое было фактически записано. Это число может быть меньше, чем было передано в аргументе *bytes*.

Важно отметить, что объекты класса FileIO находятся на самом низком уровне, образуя тонкую прослойку над системными вызовами, такими как read() и write(). В частности, при использовании этих объектов необходимо тщательно проверять возвращаемые значения, чтобы убедиться, что операции *f*.read() или *f*.write() прочитали и записали все данные. Для изменения низкоуровневых аспектов ввода-вывода, таких как блокировка файлов, поведение механизма блокировок и так далее, можно использовать модуль fcntl.

Объекты класса FileIO не должны использоваться для работы с данными, имеющими построчное представление, такими как текст. Хотя объекты и обладают методами, такими как *f*.readline() и *f*.readlines(), но они унаследованы от класса IOBase, где они реализованы исключительно на языке Python, и читают данные по одному байту за раз, используя операцию *f*.read(). Получающаяся в результате производительность просто ужасна. Например, в Python 2.6 метод *f*.readline()объекта *f* класса FileIO работает

более чем в 750 раз медленнее, чем метод *f*.readline() стандартного объекта файла, созданного функцией open().

Буферизованный ввод-вывод двоичных данных

Уровень буферизованного ввода-вывода содержит коллекцию объектов файлов, которые читают и записывают простые двоичные данные, но с буферизацией в оперативной памяти. При создании этим объектам должны передаваться объекты файлов, реализующие низкоуровневые операции ввода-вывода, такие как объекты класса FileIO, описанного в предыдущем разделе. Все классы, описываемые в этом разделе, являются производными от класса BufferedIOBase.

BufferedReader(*raw* [, *buffer_size*])

Возвращает объект класса BufferedReader, реализующий буферизованную операцию чтения двоичных данных из файла, указанного в аргументе *raw*. Аргумент *buffer_size* определяет размер буфера в байтах. Если этот аргумент опущен, используется значение по умолчанию DEFAULT_BUFFER_SIZE (8192 байтов на момент написания этих строк). Экземпляр *f* класса BufferedReader поддерживает все операции, реализованные классом IOBase, и дополнительно предоставляет реализацию следующих операций:

Метод	Описание
f.peek([*n*])	Возвращает до *n* байтов из буфера ввода-вывода. Указатель текущей позиции в файле при этом не перемещается. Если аргумент *n* не указан, возвращает единственный байт. В случае необходимости выполняется операция чтения из файла, чтобы заполнить буфер, если он пустой. Этот метод никогда не возвращает число байтов больше, чем текущий размер буфера, поэтому возвращаемый результат может содержать меньше байтов, чем было запрошено в аргументе *n*.
f.read([*n*])	Читает *n* байтов и возвращает их в виде строки байтов. При вызове без аргумента n будут прочитаны и возвращены все имеющиеся данные (до конца файла). Если файл открыт в неблокирующем режиме, будут возвращены все доступные данные. При чтении файла в неблокирующем режиме, когда нет доступных данных, возбуждается исключение BlockingIOError.
f.read1([*n*])	Читает до *n* байтов с помощью единственного обращения к системному вызову и возвращает их в виде строки байтов. Если какие-либо данные уже были загружены в буфер, они извлекаются из буфера. В противном случае производится единственный вызов метода read() низкоуровневого объекта файла. В отличие от метода *f*.read(), эта операция может возвращать меньше данных, чем было запрошено, даже если еще не был достигнут конец файла.
f.readinto(*b*)	Читает len(*b*) байтов из файла в существующий объект *b* типа bytearray. Возвращает фактическое количество прочитанных байтов. При чтении файла в неблокирующем режиме, когда нет доступных данных, возбуждается исключение BlockingIOError.

`BufferedWriter(`*raw* `[, ` *buffer_size* `[, ` *max_buffer_size*`]])`

Возвращает объект класса `BufferedWriter`, реализующий буферизованную операцию записи двоичных данных в файл, указанный в аргументе *raw*. Аргумент *buffer_size* определяет количество байтов, которые можно сохранить в буфере, прежде чем данные будут фактически записаны в поток ввода-вывода. По умолчанию используется значение `DEFAULT_BUFFER_SIZE`. В аргументе *max_buffer_size* передается максимальный размер буфера для хранения выходных данных, когда запись выполняется в неблокирующем режиме, который по умолчанию равен удвоенному значению *buffer_size*. Это значение должно быть больше, чтобы позволить продолжать запись данных, пока операционная система выполняет запись предыдущих данных из буфера в поток ввода-вывода. Экземпляр *f* класса `BufferedWriter` поддерживает следующие операции:

Метод	Описание
f.`flush()`	Записывает все содержимое буфера в поток ввода-вывода. Если вывод осуществляется в неблокирующем режиме и выполнение операции блокируется, возбуждает исключение `BlockingIOError` (например, если в текущий момент поток не может принять новую порцию данных).
f.`write(`*bytes*`)`	Записывает строку *bytes* в поток ввода-вывода и возвращает фактическое количество записанных байтов. Если вывод осуществляется в неблокирующем режиме и выполнение операции блокируется, возбуждает исключение `BlockingIOError`.

`BufferedRWPair(`*reader*`, ` *writer* `[, ` *buffer_size* `[, ` *max_buffer_size*`]])`

Возвращает объект класса `BufferedRWPair`, реализующий буферизованные операции чтения и записи двоичных данных из/в поток ввода-вывода. В аргументе *reader* передается низкоуровневый объект файла, поддерживающий операцию чтения, а в аргументе *writer* – низкоуровневый объект файла, поддерживающий операцию записи. Это могут быть разные файлы, что может оказаться удобным при организации некоторых видов взаимодействий с помощью каналов и сокетов. Аргументы *buffer_size* и *max_buffer_size* имеют тот же смысл, что и в функции `BufferedWriter()`. Экземпляры класса `BufferedRWPair` поддерживают все операции, предоставляемые объектами классов `BufferedReader` и `BufferedWriter`.

`BufferedRandom(`*raw* `[, ` *buffer_size* `[, ` *max_buffer_size*`]])`

Возвращает объект класса `BufferedRandom`, реализующий буферизованные операции чтения и записи двоичных данных из/в поток ввода-вывода, который поддерживает возможность произвольного доступа (то есть поддерживает операцию перемещения указателя текущей позиции). В аргументе *raw* должен передаваться низкоуровневый объект файла, поддерживающий операции чтения, записи и позиционирования. Аргументы *buffer_size* и *max_buffer_size* имеют тот же смысл, что и в функции `BufferedWriter()`. Экземпляры класса `BufferedRandom` поддерживают все операции, предоставляемые объектами классов `BufferedReader` и `BufferedWriter`.

`BytesIO([bytes])`

Файл, размещаемый в памяти и реализующий функциональность буферизованного потока ввода-вывода. В аргументе *bytes* передается строка байтов с начальным содержимым файла. Экземпляр *b* класса `BytesIO` поддерживает все операции, предоставляемые объектами классов `BufferedReader` и `BufferedWriter`. Кроме того, он реализует метод *b*`.getvalue()`, который возвращает текущее содержимое файла в виде строки байтов.

Как и объекты класса `FileIO`, объекты всех классов, описанных в этом разделе, не должны использоваться для работы с данными, имеющими построчное представление, такими как текст. Хотя наличие механизма буферизации немного улучшает ситуацию с производительностью, тем не менее она все еще оставляет желать лучшего (например, в Python 2.6 чтение строк выполняется более чем в 50 раз медленнее, чем при использовании объекта файла, созданного функцией `open()`). Кроме того, из-за наличия внутренней буферизации следует помнить о необходимости выполнять операцию `flush()` при записи данных. Например, если в программе используется метод *f*`.seek()` для перемещения указателя текущей позиции в новое местоположение, предварительно необходимо вызвать метод *f*`.flush()`, чтобы вытолкнуть данные, имеющиеся в буфере.

Кроме того, имейте в виду, что аргументы, определяющие размер буфера, устанавливают ограничение лишь для операций записи, что не обязательно является ограничением для использования внутренних ресурсов. Например, когда вызывается метод *f*`.write(`*data*`)` буферизованного файла *f*, все байты в строке *data* сначала будут скопированы во внутренние буферы. Если аргумент *data* представляет очень большой массив байтов, то его копирование повлечет за собой существенный расход памяти в программе. Поэтому запись данных больших объемов лучше выполнять порциями разумного размера, а не все сразу, одним вызовом операции `write()`. Следует также отметить, что модуль `io` является достаточно новым, поэтому его поведение в будущих версиях может немного измениться.

Ввод-вывод текстовой информации

Уровень ввода-вывода текстовой информации используется для работы с текстовыми данными, имеющими построчное представление. Классы, объявленные в этом разделе, построены на базе буферизованных потоков ввода-вывода и добавляют возможность работы с текстовыми строками, а также кодирование и декодирование символов Юникода. Все классы этого уровня являются производными от класса `TextIOBase`.

`TextIOWrapper(`*buffered* `[, `*encoding* `[, `*errors* `[, `*newline* `[, `*line_buffering*`]]]])`

Класс буферизованного текстового потока. В аргументе *buffered* передается объект буферизованного потока ввода-вывода, один из тех, что был описан в предыдущем разделе. В аргументе *encoding* передается строка с названием кодировки, такая как `'ascii'` или `'utf-8'`. Аргумент *errors* определяет политику обработки ошибок кодирования символов Юникода и по умолчанию имеет значение `'strict'` (описание этого аргумента приводится

в главе 9 «Ввод и вывод»). В аргументе *newline* передается последовательность символов, представляющая признак конца строки, который может принимать значения None, '', '\n', '\r' или '\r\n'. Если указано значение None, включается режим использования универсального символа перевода строки, когда при чтении любые символы конца строки преобразуются в символ '\n', а при записи символ перевода строки замещается значением os.linesep. Если в аргументе *newline* передается любое другое значение, при записи все символы '\n' будут замещаться этим значением. В аргументе *line_buffering* передается флаг, который определяет, будет ли выполняться операция flush() по окончании любой операции записи, которая выводит символ перевода строки. По умолчанию имеет значение False. Экземпляр *f* класса TextIOWrapper поддерживает все операции, реализованные классом IOBase, и дополнительно предоставляет реализацию следующих операций:

Метод	Описание
f.encoding	Имя используемой кодировки.
f.errors	Политика ошибок кодирования и декодирования.
f.line_buffering	Флаг, определяющий способ буферизации строк.
f.newlines	None, строка или кортеж со всеми различными формами преобразования символов перевода строки.
f.read([*n*])	Читает до *n* символов из потока и возвращает их в виде строки. При вызове без аргумента *n* прочитает все доступные данные до конца файла. По достижении конца файла вернет пустую строку. Возвращаемая строка декодируется в соответствии с именем кодировки в атрибуте *f*.encoding.
f.readline([*limit*])	Прочитает единственную строку текста и вернет ее. По достижении конца файла вернет пустую строку. Аргумент *limit* определяет максимальное число байтов, какое может быть прочитано.
f.write(*s*)	Записывает строку *s* в поток. Перед записью строка кодируется в соответствии с именем кодировки в атрибуте *f*.encoding.

StringIO([*initial* [, *encoding* [, *errors* [, *newline*]]]])

Возвращает объект файла, размещаемого в памяти, обладающий теми же особенностями, что и объект класса TextIOWrapper. В аргументе *initial* передается строка с начальным содержимым файла. Остальные аргументы имеют тот же смысл, что и в функции TextIOWrapper(). Экземпляр *s* класса StringIO поддерживает все обычные операции над файлами и дополнительно реализует метод *s*.getvalue(), который возвращает текущее содержимое буфера в памяти.

Функция open()

Модуль io определяет следующую функцию open(), которая является полным аналогом встроенной функции open() в Python 3:

`open(`*`file`*` [, `*`mode`*` [, `*`buffering`*` [, `*`encoding`*` [, `*`errors`*` [, `*`newline`*` [, `*`closefd`*`]]]]]])`

Открывает файл *file* и возвращает соответствующий ему объект ввода-вывода. В аргументе *file* может передаваться строка с именем файла или целочисленный дескриптор потока ввода-вывода, который уже был открыт. Результатом вызова этой функции является объект одного из классов ввода-вывода, объявленных в модуле `io`, в зависимости от значений аргументов *mode* и *buffering*. Если в аргументе *mode* указывается один из текстовых режимов, таких как `'r'`, `'w'`, `'a'` или `'U'`, возвращается экземпляр класса `TextIOWrapper`. Если в аргументе *mode* указывается двоичный режим, такой как `'rb'` или `'wb'`, тип результата зависит от значения аргумента *buffering*. Если аргумент *buffering* имеет значение 0, возвращается экземпляр класса `FileIO`, реализующий низкоуровневый ввод-вывод без буферизации. Если аргумент *buffering* имеет любое другое значение, возвращается экземпляр класса `BufferReader`, `BufferedWriter` или `BufferedRandom`, в зависимости от значения в аргументе *mode*. Аргументы *encoding*, *errors* и *newline* учитываются, только когда файл открывается в текстовом режиме; они передаются конструктору класса `TextIOWrapper`. Значение аргумента *closefd* учитывается, только если аргумент *file* представляет собой целочисленный дескриптор, и передается конструктору класса `FileIO`.

Абстрактные базовые классы

Модуль `io` объявляет следующие абстрактные базовые классы, которые можно использовать для проверки типов и объявления новых классов ввода-вывода:

Абстрактный класс	Описание
`IOBase`	Базовый класс для всех классов ввода-вывода.
`RawIOBase`	Базовый класс для объектов, поддерживающих низкоуровневый ввод-вывод двоичных данных. Является производным от класса `IOBase`.
`BufferedIOBase`	Базовый класс для объектов, поддерживающих буферизованный ввод-вывод двоичных данных. Является производным от класса `IOBase`.
`TextIOBase`	Базовый класс для объектов, поддерживающих ввод-вывод текстовых данных. Является производным от класса `IOBase`.

Программистам редко приходится работать с этими классами непосредственно. Подробное описание их использования можно найти в электронной документации.

Примечание

Модуль `io` появился совсем недавно, впервые он был включен в состав стандартной библиотеки в версии Python 3 и затем был перенесен в версию Python 2.6. К моменту написания этих строк модуль был еще недостаточно доработанным и имел очень низкую производительность, что особенно заметно в приложениях,

выполняющих массивные операции ввода-вывода текстовой информации. Если вы используете Python 2, использование встроенной функции `open()` позволит добиться лучших показателей по сравнению с использованием классов ввода-вывода, объявленных в модуле `io`. Если вы используете Python 3, то у вас нет другой альтернативы. В будущих выпусках производительность модуля наверняка будет увеличена, тем не менее многоуровневая архитектура ввода-вывода в соединении с декодированием Юникода едва ли сравнится по производительности с низкоуровневыми операциями ввода-вывода, реализованными в стандартной библиотеке языка C, которая является основой реализации механизмов ввода-вывода в Python 2.

Модуль logging

Модуль `logging` предоставляет гибкую возможность реализовать в приложениях журналирование событий, ошибок, предупреждений и отладочной информации. Эта информация может собираться, фильтроваться, записываться в файлы, отправляться в системный журнал и даже передаваться по сети на удаленные машины. В этом разделе приводятся самые основные сведения об использовании этого модуля в наиболее типичных ситуациях.

Уровни журналирования

Основной задачей модуля `logging` является получение и обработка сообщений. Каждое сообщение состоит из некоторого текста и ассоциированного с ним уровня, определяющего его важность. Уровни имеют как символические, так и числовые обозначения:

Уровень	Значение	Описание
`CRITICAL`	50	Критические ошибки/сообщения
`ERROR`	40	Ошибки
`WARNING`	30	Предупреждения
`INFO`	20	Информационные сообщения
`DEBUG`	10	Отладочная информация
`NOTSET`	0	Уровень не установлен

Эти уровни являются основой для различных функций и методов в модуле `logging`. Например, существуют методы, которые различают уровни важности сообщений и выполняют фильтрацию, блокируя запись сообщений, уровень важности которых не соответствует заданному пороговому значению.

Базовая настройка

Перед использованием функций из модуля `logging` необходимо выполнить базовую настройку специального объекта, известного как *корневой регистратор*. Корневой регистратор содержит настройки по умолчанию, включая

уровень журналирования, поток вывода, формат сообщений и другие параметры. Для выполнения настройки используется следующая функция:

`basicConfig([**kwargs])`

Выполняет базовую настройку корневого регистратора. Эта функция должна вызываться перед вызовом других функций из модуля `logging`. Она принимает множество именованных аргументов:

Именованный аргумент	Описание
`filename`	Журналируемые сообщения будут добавляться в файл с указанным именем.
`filemode`	Определяет режим открытия файла. По умолчанию используется режим `'a'` (добавление в конец).
`format`	Строка формата для формирования сообщений.
`datefmt`	Строка формата для вывода даты и времени.
`level`	Устанавливает уровень важности корневого регистратора. Обрабатываться будут сообщения с уровнем важности, равным или выше указанного. Сообщения с более низким уровнем просто будут игнорироваться.
`stream`	Определяет объект открытого файла, куда будут записываться журналируемые сообщения. По умолчанию используется поток `std.stderr`. Этот аргумент не может использоваться одновременно с аргументом `filename`.

Назначение большинства этих аргументов понятно по их названиям. Аргумент *format* используется для определения формата журналируемых сообщений с дополнительной контекстной информацией, такой как имена файлов, уровни важности, номера строк и так далее. Аргумент *datefmt* определяет формат вывода дат, совместимый с функцией `time.strftime()`. Если не определен, даты форматируются в соответствии со стандартом ISO8601.

В аргументе *format* допускается использовать следующие символы подстановки:

Формат	Описание
`%(name)s`	Имя регистратора.
`%(levelno)s`	Числовой уровень важности.
`%(levelname)s`	Символическое имя уровня важности.
`%(pathname)s`	Путь к исходному файлу, откуда была выполнена запись в журнал.
`%(filename)s`	Имя исходного файла, откуда была выполнена запись в журнал.
`%(funcName)s`	Имя функции, выполнившей запись в журнал.
`%(module)s`	Имя модуля, откуда была выполнена запись в журнал.

Формат	Описание
%(lineno)d	Номер строки, откуда была выполнена запись в журнал.
%(created)f	Время, когда была выполнена запись в журнал. Значением должно быть число, такое как возвращаемое функцией time.time().
%(asctime)s	Время в формате ASCII, когда была выполнена запись в журнал.
%(msecs)s	Миллисекунда, когда была выполнена запись в журнал.
%(thread)d	Числовой идентификатор потока выполнения.
%(threadName)s	Имя потока выполнения.
%(process)d	Числовой идентификатор процесса.
%(message)s	Текст журналируемого сообщения (определяется пользователем).

Ниже приводится пример, иллюстрирующий настройки для случая, когда в файл журнала записываются сообщения с уровнем INFO или выше:

```
import logging
logging.basicConfig(
    filename = "app.log",
    format = "%(levelname)-10s %(asctime)s %(message)s"
    level = logging.INFO
)
```

При таких настройках вывод сообщения 'Hello World' с уровнем важности CRITICAL будет выглядеть в файле журнала 'app.log', как показано ниже.

```
CRITICAL 2005-10-25 20:46:57,126 Hello World
```

Объекты класса Logger

Чтобы выводить сообщения в журнал, необходимо получить объект класса Logger. В этом разделе описывается процесс создания, настройки и использования этих объектов.

Создание экземпляра класса Logger

Создать новый объект класса Logger можно с помощью следующей функции:

```
getLogger([logname])
```

Возвращает экземпляр класса Logger с именем *logname*. Если объект с таким именем не существует, то создается и возвращается новый экземпляр класса Logger. В аргументе *logname* передается строка, определяющая имя или последовательность имен, разделенных точками (например, 'app' или 'app.net'). При вызове без аргумента *logname* вернет объект Logger корневого регистратора.

Экземпляры класса Logger создаются иначе, чем экземпляры большинства классов в других библиотечных модулях. При создании объекта Logger с помощью функции getLogger() ей всегда необходимо передавать аргумент *logname*. За кулисами функция getLogger() хранит кэш экземпляров класса Logger вместе с их именами. Если в какой-либо части программы будет запрошен регистратор с тем же именем, она возвратит экземпляр, созданный ранее. Это существенно упрощает обработку журналируемых сообщений в крупных приложениях, потому что не приходится заботиться о способах передачи экземпляров класса Logger из одного модуля программы в другой. Вместо этого в каждом модуле, где возникает необходимость журналирования сообщений, достаточно просто вызвать функцию getLogger(), чтобы получить ссылку на соответствующий объект Logger.

Выбор имен

По причинам, которые станут очевидными чуть ниже, при использовании функции getLogger() желательно всегда выбирать говорящие имена. Например, если приложение называется 'app', тогда как минимум следует использовать getLogger('app') в начале каждого модуля, составляющего приложение. Например:

```
import logging
log = logging.getLogger('app')
```

Можно также добавить имя модуля, например getLogger('app.net') или getLogger('app.user'), чтобы более четко указать источник сообщений. Реализовать это можно с помощью инструкций, как показано ниже:

```
import logging
log = logging.getLogger('app.'+__name__)
```

Добавление имен модулей упрощает выборочное отключение или перенастройку механизма журналирования для каждого модуля в отдельности, как будет описано ниже.

Запись сообщений в журнал

Если допустить, что переменная *log* является экземпляром класса Logger (созданного вызовом функции getLogger(), описанной в предыдущем разделе), то для записи сообщений с разными уровнями важности можно будет использовать следующие методы:

Уровень важности	Описание
CRITICAL	log.critical(*fmt* [, **args* [, *exc_info* [, *extra*]]])
ERROR	log.error(*fmt* [, **args* [, *exc_info* [, *extra*]]])
WARNING	log.warning(*fmt* [, **args* [, *exc_info* [, *extra*]]])
INFO	log.info(*fmt* [, **args* [, *exc_info* [, *extra*]]])
DEBUG	log.debug(*fmt* [, **args* [, *exc_info* [, *extra*]]])

В аргументе *fmt* передается строка формата, определяющая формат вывода сообщения в журнал. Все остальные аргументы в *args* будут служить параметрами спецификаторов формата в строке *fmt*. Для формирования окончательного сообщения из этих аргументов используется оператор форматирования строк %. Если передается несколько аргументов, они будут переданы оператору форматирования в виде кортежа. Если передается единственный аргумент, он будет помещен сразу вслед за оператором %. То есть, если в качестве единственного аргумента передается словарь, имеется возможность использовать имена ключей этого словаря в строке формата. Ниже приводится несколько примеров, иллюстрирующих эту возможность:

```
log = logging.getLogger("app")
# Записать сообщение, используя позиционные аргументы форматирования
log.critical("Can't connect to %s at port %d", host, port)

# Записать сообщение, используя словарь значений
parms = {
    'host' : 'www.python.org',
    'port' : 80
}
log.critical("Can't connect to %(host)s at port %(port)d", parms)
```

Если в именованном аргументе *exc_info* передается значение `True`, в сообщение добавляется информация об исключении, полученная вызовом `sys.exc_info()`. Если в этом аргументе передается кортеж с информацией об исключении, какой возвращает `sys.exc_info()`, то будет использоваться эта информация. В именованном аргументе *extra* передается словарь с дополнительными значениями для использования в строке формата (описывается ниже). Оба аргумента, *exc_info* и *extra*, должны передаваться как именованные аргументы.

Выполняя вывод журналируемых сообщений, не следует использовать возможности форматирования строк при вызове функции (то есть следует избегать приемов, когда сообщение сначала форматируется, а затем передается модулю `logging`). Например:

```
log.critical("Can't connect to %s at port %d" % (host, port))
```

В этом примере оператор форматирования строки всегда будет выполняться *перед* вызовом самой функции `log.critical()`, потому что аргументы должны передаваться функции или методу уже полностью вычисленными. Однако в более раннем примере значения для спецификаторов формата просто передаются модулю `logging` и используются, только когда сообщение действительно будет выводиться. Это очень тонкое отличие, но так как в большинстве приложений задействуется механизм фильтрации сообщений или сообщения выводятся только в процессе отладки, первый подход обеспечивает более высокую производительность, когда журналирование отключено.

В дополнение к методам, показанным выше, существует еще несколько методов экземпляра *log* класса `Logger`, позволяющих выводить сообщения в журнал.

log.exception(*fmt* [, **args*])

Выводит сообщение с уровнем ERROR и добавляет информацию о текущем обрабатываемом исключении. Может использоваться только внутри блоков except.

log.log(*level*, *fmt* [, **args* [, *exc_info* [, *extra*]]])

Выводит сообщение с уровнем *level*. Может использоваться в случаях, когда уровень важности определяется значением переменной или когда требуется использовать дополнительные уровни важности, не входящие в число пяти базовых уровней.

log.findCaller()

Возвращает кортеж (*filename*, *lineno*, *funcname*) с именем файла источника сообщения, номером строки и именем функции. Эта информация может пригодиться, например, когда желательно знать, в какой точке программы было выведено сообщение.

Фильтрование журналируемых сообщений

Каждый объект *log* класса Logger имеет свой уровень и обладает внутренним механизмом фильтрации, с помощью которого определяет, какие сообщения следует обрабатывать. Следующие два метода используются для выполнения простой фильтрации на основе числового значения уровня важности сообщений:

log.setLevel(*level*)

Устанавливает уровень важности в объекте *log*, в соответствии со значением аргумента *level*. Обрабатываться будут только сообщения с уровнем важности равным или выше значения *level*. Все остальные сообщения попросту игнорируются. По умолчанию аргумент *level* получает значение logging.NOTSET, при котором обрабатываются все сообщения.

log.isEnabledFor(*level*)

Возвращает True, если сообщение с уровнем *level* должно обрабатываться.

Журналируемые сообщения могут также фильтроваться по информации, ассоциированной с самим сообщением, например, по имени файла, по номеру строки и другим сведениям. Для этих целей используются следующие методы:

log.addFilter(*filt*)

Добавляет в регистратор объект *filt* фильтра.

log.removeFilter(*filt*)

Удаляет объект *filt* фильтра из регистратора.

Оба метода принимают в аргументе *filt* экземпляр класса Filter.

Filter(*logname*)

Создает фильтр, который пропускает только сообщения из регистратора *logname* или его потомков. Например, если в аргументе *logname* передать имя 'app', то фильтр будет пропускать сообщения из регистраторов с именами,

такими как 'app', 'app.net' или 'app.user', а сообщения из регистратора, например с именем 'spam', пропускаться не будут.

Собственные фильтры можно создавать, определяя подклассы класса Filter с собственной реализацией метода filter(*record*), который принимает запись *record* с информацией о сообщении и возвращает True или False, в зависимости от того, должно ли обрабатываться сообщение. Объект *record*, который передается этому методу, обычно обладает следующими атрибутами:

Атрибут	Описание
record.name	Имя, присвоенное объекту регистратора
record.levelname	Символическое имя уровня важности
record.levelno	Числовое значение уровня важности
record.pathname	Путь к модулю
record.filename	Имя файла модуля
record.module	Имя модуля
record.exc_info	Информация об исключении
record.lineno	Номер строки, где была выполнена попытка вывести сообщение
record.funcName	Имя функции, где была выполнена попытка вывести сообщение
record.created	Время вывода сообщения
record.thread	Числовой идентификатор потока управления
record.threadName	Имя потока управления
record.process	Числовой идентификатор текущего процесса (PID)

Следующий пример демонстрирует, как создавать собственные фильтры:

```
class FilterFunc(logging.Filter):
    def __init__(self,name):
        self.funcName = name
    def filter(self, record):
        if record.funcName == self.funcName: return False
        else: return True

log.addFilter(FilterFunc('foo')) # Игнорировать все сообщения из функции foo()
log.addFilter(FilterFunc('bar')) # Игнорировать все сообщения из функции bar()
```

Распространение сообщений и иерархии регистраторов

В приложениях со сложной организацией процесса журналирования объекты класса Logger могут объединяться в иерархии. Для этого достаточно присвоить объекту класса Logger соответствующее имя, такое как 'app.net.client'. В данном случае фактически существует три разных объекта Logger с именами 'app', 'app.net' и 'app.net.client'. Если какому-либо из реги-

страторов будет передано сообщение и оно благополучно будет пропущено фильтром этого регистратора, это сообщение продолжит распространение вверх по дереву иерархии и будет обработано всеми родительскими регистраторами. Например, сообщение, благополучно принятое регистратором 'app.net.client', будет также передано регистраторам 'app.net', 'app' и корневому регистратору.

Распространением сообщений управляют следующие атрибуты и методы объекта *log* класса Logger.

log.propagate

Логический флаг, который определяет, должно ли сообщение передаваться родительскому регистратору. По умолчанию имеет значение True.

log.getEffectiveLevel()

Возвращает **действующий уровень регистратора. Если уровень был установлен** вызовом метода setLevel(), вернет этот уровень. Если уровень не был указан явно в вызове метода setLevel() (в этом случае по умолчанию уровень принимает значение logging.NOTSET), этот метод вернет действующий уровень родительского регистратора. Если ни в одном из родительских регистраторов не был явно установлен уровень, возвращается действующий уровень корневого регистратора.

Основное назначение иерархий регистраторов состоит в том, чтобы упростить фильтрование сообщений, поступающих из различных частей крупного приложения. Например, если потребуется предотвратить вывод сообщений, поступающих из регистратора 'app.net.client', достаточно будет просто добавить следующий программный код, выполняющий необходимые настройки:

```
import logging
logging.getLogger('app.net.client').propagate = False
```

Или сохранить возможность вывода только критических сообщений:

```
import logging
logging.getLogger('app.net.client').setLevel(logging.CRITICAL)
```

Отличительной особенностью иерархий регистраторов является то обстоятельство, что решение о выводе сообщения принимается самим объектом Logger, которому это сообщение было передано, исходя из его собственного уровня важности и фильтров, а не какими-либо родительскими регистраторами. То есть, если сообщение было пропущено первым встретившимся набором фильтров, оно продолжит свое распространение и будет обработано всеми родительскими регистраторами, независимо от их фильтров и уровней важности, даже если при других обстоятельствах эти фильтры могли бы отвергнуть сообщение. На первый взгляд такой механизм работы кажется противоестественным и может даже восприниматься как ошибка. Однако установка уровня в дочернем регистраторе в более низкое значение, чем у его родителя, является одним из способов преодолеть ограничения, накладываемые родителем, и обеспечить возможность вывода сообщений с более низким уровнем важности. Например:

```
import logging
# Регистратор 'app' верхнего уровня.
log = logging.getLogger('app')
log.setLevel(logging.CRITICAL) # Принимает только сообщения с уровнем CRITICAL

# Дочерний регистратор 'app.net'
net_log = logging.getLogger('app.net')
net_log.setLevel(logging.ERROR) # Принимает сообщения с уровнем ERROR
                                # Сообщения, переданные 'app.net',
                                # теперь будут обрабатываться
                                # регистратором 'app', несмотря на то, что
                                # он имеет уровень CRITICAL.
```

При использовании иерархий регистраторов достаточно настроить только те объекты регистраторов, которые отличаются наборами фильтров или поддержкой механизма распространения сообщений. Распространение сообщения заканчивается в корневом регистраторе, поэтому он несет всю ответственность за вывод сообщения с учетом настроек, которые были выполнены с помощью функции `basicConfig()`.

Обработка сообщений

Обычно сообщения обрабатываются корневым регистратором. Однако любой объект класса `Logger` может иметь свои специальные обработчики, принимающие и обрабатывающие сообщения. Реализовать это можно с помощью следующих методов экземпляра *log* класса `Logger`.

`log.addHandler(`*`handler`*`)`

Добавляет объект класса `Handler` в регистратор.

`log.removeHandler(`*`handler`*`)`

Удаляет объект класса `Handler` из регистратора.

В модуле `logging` имеется множество различных предопределенных обработчиков, выполняющих запись сообщений в файлы, потоки, в системный журнал и так далее. Подробнее об этом рассказывается в следующем разделе. Следующий пример демонстрирует, как подключать обработчики к регистраторам с помощью указанных выше методов.

```
import logging
import sys

# Создать регистратор верхнего уровня с именем 'app'
app_log = logging.getLogger("app")
app_log.setLevel(logging.INFO)
app_log.propagate = False

# Добавить несколько обработчиков в регистратор 'app'
app_log.addHandler(logging.FileHandler('app.log'))
app_log.addHandler(logging.StreamHandler(sys.stderr))

# Отправить несколько сообщений. Они попадут в файл app.log
# и будут выведены в поток sys.stderr
app_log.critical("Creeping death detected!")
app_log.info("FYI")
```

Чаще всего добавление собственных обработчиков сообщений в регистратор выполняется с целью переопределить поведение корневого регистратора. Именно по этой причине в примере был отключен механизм распространения сообщений (то есть регистратор 'app' сам будет обрабатывать все сообщения).

Объекты класса Handler

Модуль logging предоставляет целую коллекцию предопределенных обработчиков для обработки сообщений различными способами. Эти обработчики добавляются в объекты класса Logger с помощью метода addHandler(). Кроме того, для каждого обработчика можно установить свой уровень важности и фильтры.

Встроенные обработчики

Ниже перечислены встроенные объекты обработчиков. Некоторые из них определяются в подмодуле logging.handlers, который, в случае необходимости, должен импортироваться отдельно.

handlers.DatagramHandler(*host*,*port*)

Отправляет сообщения по протоколу UDP на сервер с именем *host* и в порт *port*. Сообщения кодируются с применением соответствующего объекта словаря LogRecord и переводятся в последовательную форму с помощью модуля pickle. Сообщение, передаваемое в сеть, состоит из 4-байтового значения длины (с прямым порядком следования байтов), за которым следует упакованная запись с данными. Чтобы реконструировать сообщение, приемник должен отбросить заголовок с длиной, прочитать сообщение, распаковать его содержимое с помощью модуля pickle и вызвать функцию logging.makeLogRecord(). Так как протокол UDP **является ненадежным, ошибки** в сети могут привести к потере сообщений.

FileHandler(*filename* [, *mode* [, *encoding* [, *delay*]]])

Выводит сообщения в файл с именем *filename*. Аргумент *mode* определяет режим открытия файла и по умолчанию имеет значение 'a'. В аргументе *encoding* передается кодировка. В аргументе *delay* передается логический флаг; если он имеет значение True, открытие файла журнала откладывается до появления первого сообщения. По умолчанию имеет значение False.

handlers.HTTPHandler(*host*, *url* [, *method*])

Выгружает сообщения на сервер HTTP, используя метод HTTP GET или POST. Аргумент *host* определяет имя хоста, *url* – используемый адрес URL, а *method* – метод HTTP, который может принимать значение 'GET' (по умолчанию) или 'POST'. Сообщения кодируются с применением соответствующего объекта словаря LogRecord и преобразуются в переменные строки запроса URL с помощью функции urllib.urlencode().

handlers.MemoryHandler(*capacity* [, *flushLevel* [, *target*]])

Этот обработчик используется для сбора сообщений в памяти и периодической передачи другому обработчику, который определяется аргументом *target*. Аргумент *capacity* определяет размер буфера в байтах. В аргументе

flushLevel передается числовое значение уровня важности. Когда появляется сообщение с указанным уровнем или выше, это вынуждает обработчик передать содержимое буфера дальше. По умолчанию используется значение ERROR. В аргументе *target* передается объект класса Handler, принимающий сообщения. Если аргумент *target* опущен, вам придется определить объект-обработчик с помощью метода setTarget(), чтобы он мог выполнять обработку.

handlers.NTEventLogHandler(*appname* [, *dllname* [, *logtype*]])

В Windows NT, Windows 2000 или Windows XP отправляет сообщения в системный журнал событий. В аргументе *appname* передается имя приложения, которое будет записываться в журнал событий. В аргументе *dllname* передается полный путь к файлу .DLL или .EXE, в котором хранятся определения сообщений для журнала событий. Если аргумент опущен, *dllname* получает значение 'win32service.pyd'. В аргументе *logtype* можно передать одно из значений: 'Application', 'System' или 'Security'. По умолчанию используется значение 'Application'. Этот обработчик доступен, только если были установлены расширения Win32 для Python.

handlers.RotatingFileHandler(*filename* [, *mode* [, *maxBytes* [, *backupCount* [, *encoding* [, *delay*]]]]])

Выводит сообщение в файл *filename*. Если размер этого файла превысит значение в аргументе *maxBytes*, он будет переименован в *filename*.1 и будет открыт новый файл с именем *filename*. Аргумент *backupCount* определяет максимальное количество резервных копий файла. По умолчанию аргумент *backupCount* имеет значение 0. При любом другом значении будет выполняться циклическое переименование последовательности *filename*.1, *filename*.2, ..., *filename*.N, где *filename*.1 всегда представляет последнюю резервную копию, а *filename*.N – всегда самую старую. Режим *mode* определяет режим открытия файла журнала. По умолчанию аргумент *mode* имеет значение 'a'. Если в аргументе *maxBytes* передается значение 0 (по умолчанию), резервные копии файла журнала не создаются и размер его никак не будет ограничиваться. Аргументы *encoding* и *delay* имеют тот же смысл, что и в обработчике FileHandler.

handlers.SMTPHandler(*mailhost*, *fromaddr*, *toaddrs*, *subject* [, *credentials*])

Отправляет сообщение по электронной почте. В аргументе *mailhost* передается адрес сервера SMTP, который сможет принять сообщение. Адрес может быть простым именем хоста, указанным в виде строки, или кортежем (*host*, *port*). В аргументе *fromaddr* передается адрес отправителя, в аргументе *toaddrs* – адрес получателя и в аргументе *subject* – тема сообщения. В аргументе *credentials* передается кортеж (*username*, *password*) с именем пользователя и паролем.

handlers.SocketHandler(*host*, *port*)

Отправляет сообщение удаленному хосту по протоколу TCP. Аргументы *host* и *port* определяют адрес получателя. Сообщения отправляются в том же виде, в каком их отправляет обработчик DatagramHandler. В отличие от DatagramHandler, этот обработчик обеспечивает надежную доставку сообщений.

`StreamHandler([fileobj])`

Выводит сообщение в уже открытый объект файла *fileobj*. При вызове без аргумента сообщение выводится в поток `sys.stderr`.

`handlers.SysLogHandler([address [, facility]])`

Передает сообщение демону системного журнала в системе UNIX. В аргументе *address* передается адрес хоста назначения в виде (*host, port*). Если этот аргумент опущен, используется адрес (`'localhost', 514`). В аргументе *facility* передается целочисленный код типа источника сообщения; аргумент по умолчанию принимает значение `SysLogHandler.LOG_USER`. Полный список кодов источников сообщений можно найти в определении обработчика `SysLogHandler`.

`handlers.TimedRotatingFileHandler(filename [, when [, interval [, backupCount [, encoding [, delay [, utc]]]]]])`

То же, что и `RotatingFileHandler`, но циклическое переименование файлов происходит через определенные интервалы времени, а не по достижении файлом заданного размера. В аргументе *interval* передается число, определяющее величину интервала в единицах, а в аргументе *when* – строка, определяющая единицы измерения. Допустимыми значениями для аргумента *when* являются: `'S'` (секунды), `'M'` (минуты), `'H'` (часы), `'D'` (дни), `'W'` (недели) и `'midnight'` (ротация выполняется в полночь). Например, если в аргументе *interval* передать число `3`, а в аргументе *when* – строку `'D'`, ротация файла журнала будет выполняться каждые три дня. Аргумент *backupCount* определяет максимальное число хранимых резервных копий. В аргументе *utc* передается логический флаг, который определяет, должно ли использоваться локальное время (по умолчанию) или время по Гринвичу (UTC).

`handlers.WatchedFileHandler(filename [, mode [, encoding [, delay]]])`

То же, что и `FileHandler`, но следит за индексным узлом (inode) и устройством открытого файла журнала. Если с момента записи последнего сообщения они изменились, файл закрывается и открывается снова под тем же именем *filename*. Эти изменения могут быть вызваны удалением файла журнала или его переименованием в процессе выполнения операции ротации, произведенной внешней программой. Этот обработчик действует только в системах UNIX.

Настройка обработчиков

Каждому объекту *h* класса `Handler` может быть присвоен собственный уровень важности и фильтры. Для этого используются следующие методы:

h`.setLevel(`*level*`)`

Устанавливает порог важности обрабатываемых значений. В аргументе *level* передается числовое значение уровня, такое как `ERROR` или `CRITICAL`.

h`.addFilter(`*filt*`)`

Добавляет в обработчик объект *filt* класса `Filter`. Дополнительную информацию можно найти в описании метода `addFilter()` объектов класса `Logger`.

h.removeFilter(*filt*)

Удаляет из обработчика объект *filt* класса Filter.

Важно отметить, что уровни важности и фильтры могут устанавливаться в обработчиках независимо от объектов класса Logger, к которым подключаются обработчики. Эту особенность иллюстрирует следующий пример:

```
import logging
import sys

# Создать обработчик, который выводит сообщения с уровнем CRITICAL
# в поток stderr
crit_hand = logging.StreamHandler(sys.stderr)
crit_hand.setLevel(logging.CRITICAL)

# Создать регистратор верхнего уровня с именем 'app'
app_log = logging.getLogger("app")
app_log.setLevel(logging.INFO)
app_log.addHandler(logging.FileHandler('app.log'))
app_log.addHandler(crit_handler)
```

В этом примере создается единственный регистратор с именем 'app' и уровнем INFO. К нему подключаются два обработчика, но для одного из них (crit_handler) определен свой уровень важности CRITICAL. Хотя этот обработчик будет получать сообщения с уровнем важности INFO и выше, тем не менее он будет игнорировать сообщения, уровень которых ниже CRITICAL.

Очистка обработчиков

Следующие методы используются для выполнения завершающих операций при работе с обработчиками.

h.flush()

Выталкивает все внутренние буферы обработчика.

h.close()

Закрывает обработчик.

Форматирование сообщений

По умолчанию объекты класса Handler выводят сообщения в том виде, в каком они передаются функциям модуля logging. Однако иногда бывает необходимо добавить в сообщение дополнительную информацию, например время, имя файла, номер строки и так далее. В этом разделе рассказывается, как можно реализовать автоматическое добавление этой информации в сообщения.

Объекты форматирования

Прежде чем изменить формат сообщения, необходимо создать объект класса Formatter:

`Formatter([`*`fmt`* `[, `*`datefmt`*`]])`

Создает новый объект класса `Formatter`. В аргументе *`fmt`* передается строка формата сообщения. В строке *`fmt`* допускается использовать любые символы подстановки, перечисленные в описании функции `basicConfig()`. В аргументе *`datefmt`* передается строка форматирования дат в виде, совместимом с функцией `time.strftime()`. Если этот аргумент опущен, форматирование дат осуществляется в соответствии со стандартом ISO8601.

Чтобы задействовать объект класса `Formatter`, его необходимо подключить к обработчику. Делается это с помощью метода *`h`*`.setFormatter()` экземпляра *`h`* класса `Handler`.

`h``.setFormatter(`*`format`*`)`

Подключает объект форматирования, который будет использоваться экземпляром *`h`* класса `Handler` при создании сообщений. В аргументе *`format`* должен передаваться объект класса `Formatter`.

Следующий пример иллюстрирует, как выполняется настройка форматирования сообщений в обработчике:

```
import logging
import sys

# Определить формат сообщений
format = logging.Formatter("%(levelname)-10s %(asctime)s %(message)s")

# Создать обработчик, который выводит сообщения с уровнем CRITICAL
# в поток stderr
crit_hand = logging.StreamHandler(sys.stderr)
crit_hand.setLevel(logging.CRITICAL)
crit_hand.setFormatter(format)
```

В этом примере нестандартный объект форматирования подключается к обработчику `crit_hand`. Если этому обработчику передать сообщение, такое как `"Creeping death detected."`, он выведет следующий текст:

```
CRITICAL 2005-10-25 20:46:57,126 Creeping death detected.
```

Добавление в сообщения дополнительной контекстной информации

В некоторых случаях бывает полезно добавлять в журналируемые сообщения дополнительную контекстную информацию. Эта информация может быть получена одним из двух способов. Во-первых, все основные операции журналирования (такие как *`log`*`.critical()`, *`log`*`.warning()` и другие) принимают именованный аргумент *`extra`*, в котором можно передать словарь с дополнительными полями для использования в строках формата. Эти поля объединяются с контекстными данными объектов `Formatter`. Например:

```
import logging, socket
logging.basicConfig(
    format = "%(hostname)s %(levelname)-10s %(asctime)s %(message)s"
)
```

```
# Дополнительные контекстные данные
netinfo = {
    'hostname' : socket.gethostname(),
    'ip'       : socket.gethostbyname(socket.gethostname())
}
log = logging.getLogger('app')

# Вывести сообщение с дополнительными контекстными данными
log.critical("Could not connect to server", extra=netinfo)
```

Недостатком такого подхода является необходимость передавать дополнительную информацию каждой операции журналирования, в противном случае программа будет завершаться аварийно. Другой подход основан на использовании класса `LogAdapter` в качестве обертки для существующего регистратора.

`LogAdapter(`*`log`* `[,` *`extra`*`])`

Создает обертку вокруг объекта *log* класса `Logger`. В аргументе *extra* передается словарь с дополнительной контекстной информацией, которая передается объекту форматирования. Экземпляры класса `LogAdapter` имеют тот же интерфейс, что и объекты класса `Logger`. Однако методы, которым передаются сообщения, автоматически будут добавлять дополнительную информацию, полученную в аргументе *extra*.

Следующий пример демонстрирует использование объекта `LogAdapter`:

```
import logging, socket
logging.basicConfig(
    format = "%(hostname)s %(levelname)-10s %(asctime)s %(message)s"
)

# Дополнительные контекстные данные
netinfo = {
    'hostname' : socket.gethostname(),
    'ip'       : socket.gethostbyname(socket.gethostname())
}

# Создать регистратор
log = logging.LogAdapter(logging.getLogger("app"), netinfo)

# Вывести сообщение.
# Дополнительные контекстные данные поставляются объектом LogAdapter
log.critical("Could not connect to server")
```

Различные вспомогательные функции

Следующие функции из модуля `logging` управляют различными параметрами журналирования:

`disable(`*`level`*`)`

На глобальном уровне запрещает вывод сообщений с уровнем важности ниже значения *level*. Эта функция может использоваться для отключения журналирования для всего приложения, например, если временно нужно запретить вывод или уменьшить поток журналируемых сообщений.

`addLevelName(level, levelName)`

Создает новый уровень важности с указанным именем. В аргументе *level* передается числовое значение, а в аргументе *levelName* – строка. Может использоваться для изменения имен встроенных уровней или для добавления новых уровней, вдобавок к поддерживаемым по умолчанию.

`getLevelName(level)`

Возвращает имя уровня, соответствующего числовому значению *level*.

`shutdown()`

Уничтожает все объекты регистраторов, предварительно выталкивает буферы, если необходимо.

Настройка механизма журналирования

Настройка приложения на использование модуля `logging` обычно выполняется в несколько основных этапов:

1. С помощью функции `getLogger()` создается несколько объектов класса `Logger`. Соответствующим образом устанавливаются значения параметров, таких как уровень важности.
2. Создаются объекты обработчиков различных типов (таких как `FileHandler`, `StreamHandler`, `SocketHandler` и так далее) и устанавливаются соответствующие уровни важности.
3. Создаются объекты класса `Formatter` и подключаются к объектам `Handler` с помощью метода `setFormatter()`.
4. С помощью метода `addHandler()` объекты `Handler` подключаются к объектам `Logger`.

Каждый этап может оказаться достаточно сложным, поэтому лучше всего поместить реализацию настройки механизма журналирования в одном, хорошо документированном, месте. Например, можно создать файл `applogconfig.py`, который будет импортироваться основным модулем приложения:

```
# applogconfig.py
import logging
import sys

# Определить формат сообщений
format = logging.Formatter("%(levelname)-10s %(asctime)s %(message)s")

# Создать обработчик, который выводит сообщения с уровнем CRITICAL
# в поток stderr
crit_hand = logging.StreamHandler(sys.stderr)
crit_hand.setLevel(logging.CRITICAL)
crit_hand.setFormatter(format)

# Создать обработчик, который выводит сообщения в файл
applog_hand = logging.FileHandler('app.log')
applog_hand.setFormatter(format)

# Создать регистратор верхнего уровня с именем 'app'
app_log = logging.getLogger("app")
```

```
app_log.setLevel(logging.INFO)
app_log.addHandler(applog_hand)
app_log.addHandler(crit_hand)

# Изменить уровень важности для регистратора 'app.net'
logging.getLogger("app.net").setLevel(logging.ERROR)
```

Если в какой-либо части процедуры настройки потребуется что-то изменить, легче будет учесть все нюансы, если вся процедура будет реализована в одном месте. Имейте в виду, что этот специальный файл должен импортироваться только один раз и только в одном месте программы. В других модулях программы, где потребуется выводить журналируемые сообщения, достаточно просто добавить следующие строки:

```
import logging
app_log = logging.getLogger("app")
...
app_log.critical("An error occurred")
```

Подмодуль logging.config

Помимо жесткого определения настроек механизма журналирования в программном коде на языке Python имеется также возможность реализовать настройку модуля `logging` в виде файла с настройками в формате INI. Для этого используются следующие функции из модуля `logging.config`.

`fileConfig(`*`filename`* `[,` *`defaults`* `[,` *`disable_existing_loggers`*`]])`

Читает настройки механизма журналирования из файла с именем *`filename`*. В аргументе *`defaults`* передается словарь со значениями параметров по умолчанию для использования в файле с настройками. Содержимое файла *`filename`* читается с помощью модуля `ConfigParser`. В аргументе *`disable_existing_loggers`* передается логический флаг, который определяет, должны ли отключаться существующие регистраторы после чтения новых параметров настройки. По умолчанию имеет значение `True`.

Дополнительную информацию, касающуюся ожидаемого формата файлов с настройками, можно найти в электронной документации к модулю `logging`. Однако опытные программисты наверняка смогут понять основные принципы из следующего примера, который является версией файла с настройками, аналогичными тем, что выполнялись в примере модуля `applogconfig.py` выше.

```
; applogconfig.ini
;
; Файл с настройками модуля logging
; В следующих разделах определяются имена для объектов Logger, Handler
; и Formatter, которые настраиваются ниже.
[loggers]
keys=root,app,app_net

[handlers]
keys=crit,applog

[formatters]
```

```
keys=format

[logger_root]
level=NOTSET
handlers=

[logger_app]
level=INFO
propagate=0
qualname=app
handlers=crit,applog

[logger_app_net]
level=ERROR
propagate=1
qualname=app.net
handlers=

[handler_crit]
class=StreamHandler
level=CRITICAL
formatter=format
args=(sys.stderr,)

[handler_applog]
class=FileHandler
level=NOTSET
formatter=format
args=('app.log',)

[formatter_format]
format=%(levelname)-10s %(asctime)s %(message)s
datefmt=
```

Для чтения содержимого этого файла и настройки механизма журналирования можно использовать следующий фрагмент программного кода:

```
import logging.config
logging.config.fileConfig('applogconfig.ini')
```

Как и ранее, нет никакой необходимости беспокоиться о настройке механизма журналирования в модулях, где потребуется выводить журналируемые сообщения. В них достаточно просто импортировать модуль `logging` и получить ссылку на соответствующий объект `Logger`. Например:

```
import logging
app_log = logging.getLogger("app")
...
app_log.critical("An error occurred")
```

Вопросы производительности

Добавление в приложение механизма журналирования может существенно ухудшить его производительность, если не отнестись к этому с должным вниманием. Однако существуют некоторые приемы, которые могут помочь уменьшить это отрицательное влияние.

Во-первых, при компиляции в оптимизированном режиме (-O) удаляется весь программный код, который выполняется в условных инструкциях, таких как `if __debug__: инструкции`. Если отладка – единственная цель использования модуля `logging`, можно поместить все вызовы механизма журналирования в условные инструкции, которые автоматически будут удаляться при компиляции в оптимизированном режиме.

Второй прием заключается в использовании «пустого» объекта `Null` вместо объектов `Logger`, когда журналирование должно быть полностью отключено. Этот прием отличается от использования `None` тем, что основан на использовании объектов, которые просто пропускают все обращения к ним. Например:

```
class Null(object):
    def __init__(self, *args, **kwargs): pass
    def __call__(self, *args, **kwargs): return self
    def __getattribute__(self, name): return self
    def __setattr__(self, name, value): pass
    def __delattr__(self,name): pass

log = Null()
log.critical("An error occurred.") # Ничего не делает
```

При должном уровне подготовленности журналированием можно также управлять с помощью декораторов и метаклассов. Поскольку эти две особенности потребляют время только на этапе, когда Python интерпретирует определения функций, методов и классов, они позволяют добавлять и удалять поддержку журналирования в различных частях программы и избавиться от потери производительности, когда журналирование выключено. Дополнительная информация по этой теме приводится в главе 6 «Функции и функциональное программирование» и в главе 7 «Классы и объектно-ориентированное программирование».

Примечания

- Модуль `logging` имеет множество параметров настройки, которые не обсуждались здесь. За дополнительными подробностями читателям следует обращаться к электронной документации.
- Модуль `logging` может использоваться в многопоточных программах. В частности, нет никакой необходимости окружать операциями блокировки программный код, который выводит журналируемые сообщения.

Модуль mmap

Модуль `mmap` обеспечивает поддержку объектов отображений файлов в памяти. Эти объекты ведут себя, как обычные файлы и строки байтов, и в большинстве случаев могут использоваться вместо обычных файлов и строк байтов. Кроме того, содержимое отображений файлов в памяти является изменяемым. Это означает, что любые изменения могут быть выполнены с помощью операций индексирования и получения срезов. Если

отображение файла не было сделано частным, такие изменения напрямую будут отражаться на содержимом самого файла.

Отображение файла в памяти создается с помощью функции `mmap()`, которая немного отличается в версиях для UNIX и Windows.

`mmap(`*`fileno`*`, `*`length`*` [, `*`flags`*`, [`*`prot`*` [,`*`access`*` [, `*`offset`*`]]]])`

(UNIX). Возвращает объект `mmap` и отображает *`length`* байтов из файла, определяемого целочисленным дескриптором *`fileno`*. Если в аргументе *`fileno`* передать значение `-1`, будет создано анонимное отображение памяти. Аргумент *`flags`* определяет природу отображения и может принимать одно из следующих значений:

Флаг	Описание
`MAP_PRIVATE`	Создает частную копию отображения. Изменения в объекте будут наблюдаться только в данном процессе.
`MAP_SHARED`	Создает отображение, которое может совместно использоваться всеми процессами, которые отображают одну и ту же область файла. Изменения в объекте будут оказывать влияние на все отображения.

По умолчанию аргумент *`flags`* имеет значение `MAP_SHARED`. Аргумент *`prot`* определяет уровень защиты памяти объекта и представляет собой битовую маску, составленную из следующих флагов с помощью битовой операции ИЛИ:

Флаг	Описание
`PROT_READ`	Данные в объекте доступны для чтения.
`PROT_WRITE`	Объект позволяет изменять содержимое.
`PROT_EXEC`	Объект может содержать выполняемые инструкции.

По умолчанию аргумент *`prot`* получает значение `PROT_READ | PROT_WRITE`. Флаги, указанные в аргументе *`prot`*, должны соответствовать режиму открытия дескриптора файла *`fileno`*. В большинстве случаев это означает, что файл должен быть открыт для чтения/записи (например, `os.open(name, os.O_RDWR)`).

Необязательный аргумент *`access`* может использоваться в качестве альтернативы аргументам *`flags`* и *`prot`*. Этот аргумент может принимать одно из следующих значений:

Права доступа	Описание
`ACCESS_READ`	Объект доступен только для чтения.
`ACCESS_WRITE`	Объект доступен для чтения/записи со сквозной записью. Изменения немедленно записываются в сам файл.
`ACCESS_COPY`	Объект доступен для чтения/записи с копированием при записи. Изменения могут выполняться в памяти, но они не записываются в сам файл.

Обычно аргумент *access* передается в виде именованного аргумента, например mmap(*fileno*, *length*, access=ACCESS_READ). Одновременная передача аргументов *access* и *flags* считается ошибкой. Аргумент *offset* определяет смещение отображаемой области в байтах относительно начала файла и по умолчанию принимает значение 0. Это значение должно быть кратным значению mmap.ALLOCATIONGRANULARITY.

mmap(*fileno*, *length*[, *tagname* [, *access* [, *offset*]]])

(Windows) Возвращает объект mmap и отображает *length* байтов из файла, определяемого целочисленным дескриптором *fileno*. Если в аргументе *fileno* передать значение -1, будет создано анонимное отображение памяти. Если в аргументе *length* передать значение, которое превышает размер файла, файл будет увеличен до размера *length*. Если в аргументе *length* передать значение 0, то при создании отображения будет использоваться текущая длина файла, если файл не пустой (в противном случае будет возбуждено исключение). В необязательном аргументе *tagname* можно передать строку, которую можно будет использовать в качестве имени отображения. Если в аргументе *tagname* передать имя существующего отображения, будет открыто это отображение. В противном случае будет создано новое отображение. Если в аргументе *tagname* передать значение None, будет создано неименованное отображение. Необязательный аргумент *access* определяет режим доступа. Он может принимать те же значения, что были перечислены выше, в описании версии mmap() для UNIX. По умолчанию аргумент *access* принимает значение ACCESS_WRITE. Аргумент *offset* определяет смещение отображаемой области в байтах относительно начала файла и по умолчанию принимает значение 0. Это значение должно быть кратным значению mmap.ALLOCATIONGRANULARITY.

Объект *m* отображения файла в памяти поддерживает следующие методы.

m.close()

Закрывает файл. Последующие попытки выполнить какие-либо операции будут вызывать исключение.

m.find(*string*[, *start*])

Возвращает индекс первого вхождения строки *string*. Необязательный аргумент *start* определяет позицию начала поиска. Возвращает -1, если искомая строка не была найдена.

m.flush([*offset*, *size*])

Записывает изменения, выполненные в памяти, обратно в файл. Аргументы *offset* и *size* определяют диапазон участка памяти, который должен быть записан. В противном случае выполняется запись всего отображения.

m.move(*dst*,*src*,*count*)

Копирует *count* байтов, начиная с позиции *src*, в позицию *dst*. Копирование выполняется с помощью функции memmove() языка C, которая гарантирует корректную работу, когда исходная и конечная области копирования перекрываются.

m.read(*n*)

Читает до *n* байтов, начиная с текущей позиции в файле, и возвращает данные в виде строки.

m.read_byte()

Читает единственный байт, начиная с текущей позиции в файле, и возвращает данные в виде строки длины 1.

m.readline()

Возвращает строку текста, начиная с текущей позиции в файле.

m.resize(*newsize*)

Изменяет размер объекта отображаемой памяти до нового размера *newsize* в байтах.

m.seek(*pos*[, *whence*])

Изменяет местоположение текущей позиции в файле. Аргументы *pos* и *whence* имеют тот же смысл, что и в методе seek() объектов файлов.

m.size()

Возвращает длину файла. Это значение может быть больше, чем размер отображаемой области.

m.tell()

Возвращает значение текущей позиции в файле.

m.write(*string*)

Записывает строку байтов в файл, начиная с текущей позиции.

m.write_byte(*byte*)

Записывает единственный байт в память, в текущую позицию.

Примечания

- Несмотря на то что для **UNIX и Windows предоставляются немного отличающиеся версии** функции mmap(), тем не менее этот модуль позволяет писать переносимый программный код, опирающийся на необязательный аргумент *access*, который является общим для обеих версий функции. Например, вызов mmap(*fileno*,*length*,access=ACCESS_WRITE) будет работать как в UNIX, так и в Windows.
- Некоторые отображения в память можно создать, только если их длина кратна размеру страницы памяти в системе, который хранится в виде константы mmap.PAGESIZE.
- В системах **UNIX SVR4 анонимное отображение в память можно получить** вызовом mmap() для файла /dev/zero, открытого с соответствующими разрешениями.
- В системах **UNIX BSD анонимное отображение в память можно получить** вызовом mmap() с отрицательным дескриптором файла и флагом mmap.MAP_ANON.

Модуль msvcrt

Модуль msvcrt предоставляет доступ к множеству интересных функций из библиотеки времени выполнения Microsoft Visual C. Этот модуль доступен только в Windows.

getch()

Считывает нажатую клавишу и возвращает соответствующий символ. Этот вызов блокируется в ожидании нажатия клавиши. Если нажата специальная функциональная клавиша, функция возвращает '\000' или '\xe0', а следующий ее вызов возвращает код клавиши. Эта функция не выводит эхо-символы в консоль и не может использоваться для чтения комбинации Ctrl+C.

getwch()

То же, что и getch(), за исключением того, что возвращает символ Юникода.

getche()

То же, что и getch(), за исключением того, что выводит эхо-символ (если он печатаемый).

getwche()

То же, что и getche(), за исключением того, что возвращает символ Юникода.

get_osfhandle(*fd*)

Возвращает числовой идентификатор для дескриптора файла *fd*. Возбуждает исключение IOError, если *fd* не соответствует известному дескриптору файла.

heapmin()

Принуждает внутреннего диспетчера памяти Python вернуть неиспользуемые блоки операционной системе. Работает только в Windows NT и в случае ошибки возбуждает исключение IOError.

kbhit()

Возвращает True, если во входном буфере клавиатуры присутствуют доступные для чтения данные.

locking(*fd*, *mode*, *nbytes*)

Блокирует часть файла, соответствующего дескриптору *fd*, созданному средствами библиотеки времени выполнения языка C. В аргументе *nbytes* передается протяженность блокируемой области в байтах, начиная от текущей позиции в файле. Аргумент *mode* может принимать одно из следующих целочисленных значений:

Значение	Описание
0	Разблокирует область файла (LK_UNLCK)
1	Блокирует область файла (LK_LOCK)

(продолжение)

Значение	Описание
2	Блокирует область файла. Устанавливает неблокирующий режим (LK_NBLCK)
3	Блокирует область файла для записи (LK_RLCK)
4	Блокирует область файла для записи. Устанавливает неблокирующий режим (LK_NBRLCK)

Попытка удерживать блокировку больше 10 секунд приведет к исключению IOError.

`open_osfhandle(`*`handle`*`, `*`flags`*`)`

Создает дескриптор файла средствами библиотеки времени выполнения языка C из числового идентификатора *handle*. В аргументе *flags* передается битная маска, составленная из значений os.O_APPEND, os.O_RDONLY и os.O_TEXT, объединенных битовой операцией ИЛИ. Возвращает целочисленный файловый дескриптор, который можно передать функции os.fdopen() для создания объекта файла.

`putch(`*`char`*`)`

Выводит символ *char* в консоль без буферизации.

`putwch(`*`char`*`)`

То же, что и putch(), за исключением того, что в аргументе *char* передается символ Юникода.

`setmode(`*`fd`*`, `*`flags`*`)`

Устанавливает режим преобразования символов завершения строки для дескриптора файла *fd*. Аргумент *flags* может принимать значения os.O_TEXT — для текстового режима и os.O_BINARY — для двоичного режима.

`ungetch(`*`char`*`)`

«Возвращает» символ обратно в буфер консоли. Этот символ будет первым символом, доступным функциям getch() и getche().

`ungetwch(`*`char`*`)`

То же, что и ungetch(), за исключением того, что в аргументе *char* передается символ Юникода.

Примечание

Существует большое разнообразие расширений для платформы Win32, обеспечивающих доступ к библиотеке Microsoft Foundation Classes, компонентам COM, визуальным элементам графического интерфейса пользователя и так далее. Эти темы выходят далеко за рамки данной книги, однако подробное описание многих из них можно найти в книге Марка Хаммонда (Mark Hammond) и Энди Робинсона (Andy Robinson) «Python Programming on Win32» (O'Reilly & Associates, 2000). Кроме того, на сайте *http://www.python.org* можно найти обширный список сторонних модулей для использования на платформе Windows.

См. также

Описание модуля winreg (стр. 511).

Модуль optparse

Модуль optparse обеспечивает высокоуровневую поддержку обработки параметров командной строки в стиле UNIX, которые передаются в переменной sys.argv. Простой пример использования этого модуля приводится в главе 9. Использование модуля optparse в основном сводится к использованию класса OptionParser.

```
OptionParser([**args])
```

Создает и возвращает новый анализатор параметров командной строки – экземпляр класса OptionParser. Может принимать самые разные именованные аргументы, определяющие настройки анализатора. Эти именованные аргументы описаны в следующем списке:

Именованный аргумент	Описание
add_help_option	Указывает, поддерживается ли специальный параметр вывода справки (--help и -h). По умолчанию получает значение True.
conflict_handler	Определяет порядок обработки конфликтующих параметров командной строки. Может принимать значение 'error' (по умолчанию) или 'resolve'. Если указано значение 'error', при добавлении в анализатор конфликтующих строк параметров возбуждается исключение optparse.OptionConflictError. Если указано значение 'resolve', конфликты разрешаются так, что параметр, добавленный позднее, имеет более высокий приоритет. При этом ранее добавленные параметры также могут оставаться доступными, если они были добавлены под несколькими именами, из которых хотя бы для одного не обнаруживается конфликтов.
description	Строка с описанием, которая отображается при выводе справки. Эта строка автоматически форматируется так, чтобы уместиться по ширине экрана.
formatter	Экземпляр класса optparse.HelpFormatter, используемый для форматирования текста при выводе справки. Может иметь значение optparse.IndentedHelpFormatter (по умолчанию) или optparse.TitledHelpFormatter.
option_class	Класс на языке Python, который используется для хранения информации о каждом параметре командной строки. По умолчанию используется класс optparse.Option.
option_list	Список параметров, используемый для заполнения анализатора. По умолчанию этот список пуст, а добавление параметров осуществляется с помощью метода add_option(). В противном случае этот список должен содержать объекты типа Option.

(продолжение)

Именованный аргумент	Описание
prog	Имя программы, которое будет использоваться вместо '%prog' в тексте справки.
usage	Строка со справочной информацией об использовании приложения, которая выводится в ответ на использование параметра --help или при встрече недопустимого параметра. По умолчанию используется строка '%prog [options]', где элемент '%prog' замещается либо значением, возвращаемым функцией os.path.basename(sys.argv[0]), либо значением именованного аргумента prog (если было указано). Чтобы подавить вывод сообщения об использовании приложения, можно передать значение optparse.SUPPRESS_USAGE.
version	Строка с номером версии, которая выводится в ответ на использование параметра --version. По умолчанию получает значение None, а параметр --version не добавляется в анализатор. Когда в этом аргументе указывается иное значение, параметр --version добавляется автоматически. В строке может присутствовать специальный элемент '%prog', который будет замещен именем программы.

Когда не требуется выполнять нестандартную настройку обработки параметров командной строки, экземпляр класса OptionParser обычно создается вызовом функции без аргументов. Например:

```
p = optparse.OptionParser()
```

Экземпляр *p* класса OptionParser поддерживает следующие методы:

p.add_option(*name1*, ..., *nameN* [, ***parms*])

Добавляет новый параметр в объект *p*. Аргументы *name1*, *name2* и так далее представляют различные имена параметров. Например, можно указывать короткие и длинные имена параметров, такие как '-f' и '-file'. Вслед за именами параметров могут следовать необязательные именованные аргументы, которые определяют порядок обработки параметров при их встрече в командной строке. Эти именованные аргументы описаны в следующем списке:

Именованный аргумент	Описание
action	Действие, выполняемое при анализе параметра. Допустимыми являются следующие значения: 'store' – требуется прочитать и сохранить аргумент параметра. Это значение используется по умолчанию, если явно не было указано никакого другого. 'store_const' – параметр не имеет аргументов, но когда он встречается, требуется сохранить значение константы, указанное в именованном аргументе const.

Именованный аргумент	Описание
	'store_true' – действует так же, как и 'store_const', но при встрече параметра сохраняет логическое значение True. 'store_false' – действует так же, как и 'store_true', но при встрече параметра сохраняет логическое значение False. 'append' – параметр имеет аргумент, который должен быть добавлен в конец списка, когда он встречается. Применяется, когда в командной строке используется несколько одинаковых параметров, с целью обеспечить возможность передачи нескольких значений. 'count' – параметр не имеет аргументов, но когда он встречается, требуется сохранить значение счетчика. Значение счетчика увеличивается на 1 всякий раз, когда встречается один и тот же параметр. 'callback' – при встрече параметра требуется вызвать функцию обратного вызова, указанную в именованном аргументе callback. 'help' – при встрече параметра требуется вывести справочное сообщение. Это необходимо, только если требуется обеспечить вывод справочной информации с помощью другого параметра, отличного от стандартных -h и --help. 'version' – при встрече параметра требуется вывести номер версии, переданный функции OptionParser(). Это необходимо, только если требуется обеспечить вывод номера версии с помощью другого параметра, отличного от стандартных -v и --version.
callback	Определяет функцию обратного вызова, которая должна вызываться при встрече параметра. В качестве функции можно использовать любой вызываемый объект Python, который вызывается как callback(option, opt_str, value, parser, *args, **kwargs). В аргументе *option* передается экземпляр класса optparse.Option, в аргументе *opt_str* – строка с параметром из командной строки, который вызвал обращение к функции, в аргументе *value* – значение параметра (если имеется), в аргументе *parser* – действующий экземпляр класса OptionParser, в аргументе *args* – позиционные аргументы, переданные в именованном аргументе callback_args, и в аргументе *kwargs* – именованные аргументы, переданные в именованном аргументе callback_kwargs.
callback_args	Необязательный позиционный аргумент, который должен передаваться функции, указанной в аргументе callback.
callback_kwargs	Необязательный именованный аргумент, который должен передаваться функции, указанной в аргументе callback.
choices	Список строк, которые определяют все возможные значения параметра. Используется, только когда параметр может принимать ограниченный набор значений аргумента (например, ['small', 'medium', 'large']).

(продолжение)

Именованный аргумент	Описание
const	Значение константы, которое сохраняется действием 'store_const'.
default	Устанавливает значение параметра по умолчанию, если он отсутствует в командной строке. По умолчанию используется значение None.
dest	Определяет имя атрибута, который будет использоваться для сохранения значения параметра командной строки. Обычно используется имя, соответствующее имени самого параметра.
help	Пояснительный текст для данного параметра. Если не указан, при выводе справки параметр будет перечислен без дополнительного описания. Для сокрытия параметра можно использовать значение optparse.SUPPRESS_HELP. В этом тексте может присутствовать специальный элемент '%default', который будет замещен значением аргумента *default*.
metavar	Определяет имя аргумента параметра, которое будет использоваться при выводе справочного текста.
nargs	Определяет количество аргументов параметра для действий, которые предполагают наличие аргументов. По умолчанию принимает значение 1. Если это число больше 1, аргументы параметра будут собраны в кортеж, который затем можно будет использовать при обработке.
type	Определяет тип параметра. Допустимыми являются следующие типы: 'string' (по умолчанию), 'int', 'long', 'choice', 'float' и 'complex'.

p.disable_interspersed_args()

Запрещает смешивание простых параметров с позиционными аргументами. Например, если '-x' и '-y' являются простыми параметрами, не имеющими аргументов, эти параметры должны указываться перед любыми аргументами (например: 'prog -x -y arg1 arg2 arg3').

p.enable_interspersed_args()

Разрешает смешивание простых параметров с позиционными аргументами. Например, если '-x' и '-y' являются простыми параметрами, не имеющими аргументов, они могут смешиваться с позиционными аргументами, например: 'prog -x arg1 arg2 -y arg3'. Это поведение по умолчанию.

p.parse_args([*arglist*])

Анализирует параметры командной строки и возвращает кортеж (*options*, *args*), где *options* – это объект, содержащий все параметры, а *args* – список всех оставшихся позиционных аргументов. Объект *options* хранит все данные параметров в атрибутах с именами, соответствующими именам параметров. Например, информация о параметре '--output' будет храниться в атрибуте *options*.output. Если параметр не был указан в командной стро-

ке, его значением будет None. Имя атрибута может быть определено с помощью именованного аргумента dest в вызове метода add_option(), описанного выше. По умолчанию аргументы извлекаются из sys.argv[1:]. Однако в необязательном аргументе *arglist* можно определить другой источник аргументов командной строки.

p.set_defaults(*dest*=*value*, ... *dest*=*value*)

Устанавливает значения по умолчанию для указанных параметров. Методу просто нужно передать именованные аргументы, определяющие параметры, для которых устанавливаются значения по умолчанию. Имена аргументов должны совпадать с именами, указанными в аргументе *dest* метода add_option(), описанного выше.

p.set_usage(*usage*)

Изменяет строку с текстом, описывающим порядок использования приложения, который выводится при использовании параметра --help.

Пример

```
# foo.py
import optparse
p = optparse.OptionParser()

# Простой параметр без аргументов
p.add_option("-t", action="store_true", dest="tracing")

# Параметр, принимающий строковый аргумент
p.add_option("-o", "--outfile", action="store", type="string", dest="outfile")

# Параметр, принимающий целочисленный аргумент
p.add_option("-d", "--debuglevel", action="store", type="int", dest="debug")

# Параметр с небольшим числом допустимых значений аргумента
p.add_option("--speed", action="store", type="choice", dest="speed",
             choices=["slow","fast","ludicrous"])

# Параметр, принимающий несколько аргументов
p.add_option("--coord", action="store", type="int", dest="coord", nargs=2)

# Группа параметров, значения которых сохраняются в одном и том же атрибуте
p.add_option("--novice", action="store_const", const="novice", dest="mode")
p.add_option("--guru", action="store_const", const="guru", dest="mode")

# Установить значения по умолчанию для различных параметров
p.set_defaults(tracing=False,
               debug=0,
               speed="fast",
               coord=(0,0),
               mode="novice")

# Проанализировать аргументы
opt, args = p.parse_args()

# Вывести значения параметров
print "tracing :", opt.tracing
```

```
print “outfile :”, opt.outfile
print “debug   :”, opt.debug
print “speed   :”, opt.speed
print “coord   :”, opt.coord
print “mode    :”, opt.mode

# Вывести оставшиеся аргументы
print “args    :”, args
```

Ниже приводится пример короткого интерактивного сеанса в системе UNIX, который демонстрирует работу предыдущего программного кода:

```
% python foo.py -h
usage: foo.py [options]

options:
  -h, --help show this help message and exit
  -t
  -o OUTFILE, --outfile=OUTFILE
  -d DEBUG, --debuglevel=DEBUG
  --speed=SPEED
  --coord=COORD
  --novice
  --guru

% python foo.py -t -o outfile.dat -d 3 --coord 3 4 --speed=ludicrous blah
tracing : True
outfile : outfile.dat
debug   : 3
speed   : ludicrous
coord   : (3, 4)
mode    : novice
args    : [‘blah’]

% python foo.py --speed=insane
usage: foo.py [options]

foo.py:error:option --speed:invalid choice:’insane’
(choose from ‘slow’, ‘fast’, ‘ludicrous’)
```

Примечания

- При **определении имен параметров используйте одиночный символ дефиса** для определения коротких имен, таких как ‘-x’, и двойной символ дефиса для определения длинных имен, таких как ‘--exclude’. **Если попытаться определить параметр, в котором смешиваются оба стиля, например:** ‘-exclude’, **будет возбуждено исключение** OptionError.
- В стандартной библиотеке языка Python имеется также модуль getopt, который обеспечивает поддержку разбора параметров командной строки в стиле библиотеки языка C с тем же именем. Однако он не имеет никаких практических преимуществ перед модулем optparse (который находится на более высоком уровне, и программный код с его использованием получается компактнее).

- Модуль optparse обладает массой дополнительных возможностей по настройке и специализированной обработке определенных видов параметров командной строки. Однако эти возможности не являются необходимыми в типичных случаях анализа параметров командной строки. Дополнительные сведения и примеры использования можно найти в электронной документации.

Модуль os

Модуль os предоставляет переносимый интерфейс к часто используемым службам операционной системы. Для этого он отыскивает встроенный модуль, зависящий от типа операционной системы, такой как nt или posix, и экспортирует функции и данные, находящиеся в нем. Функции, перечисленные ниже, доступны в Windows и UNIX, если явно не оговаривается иное. Под системой UNIX здесь также подразумеваются системы Linux и Mac OS X.

В модуле os объявлены следующие переменные общего назначения:

`environ`

Объект отображения, представляющий текущие переменные окружения. Изменения в этом отображении отражаются на текущем окружении. Если доступна функция putenv(), то изменения также отражаются на окружении дочерних процессов.

`linesep`

Строка, используемая в текущей платформе для отделения строк в тексте. Может состоять из единственного символа, такого как '\n' в системах POSIX, или из нескольких, например: '\r\n' в Windows.

`name`

Имя импортируемого модуля, в зависимости от типа платформы: 'posix', 'nt', 'dos', 'mac', 'ce', 'java', 'os2' или 'riscos'.

`path`

Имя стандартного модуля, используемого для работы с путями в файловой системе, в зависимости от типа платформы. Этот модуль можно также загрузить с помощью инструкции import os.path.

Окружение процесса

Ниже перечислены функции, которые используются для получения и изменения различных параметров окружения, в котором выполняется процесс. Идентификаторы процесса, группы, группы процессов и сеанса являются целыми числами, если не оговаривается иное.

`chdir(`*`path`*`)`

Назначает *path* текущим рабочим каталогом.

`chroot(`*`path`*`)`

Изменяет корневой каталог для текущего процесса (UNIX).

`ctermid()`

Возвращает имя файла управляющего терминала для процесса (UNIX).

`fchdir(fd)`

Изменяет текущий рабочий каталог. В аргументе *fd* передается дескриптор файла открытого каталога (UNIX).

`getcwd()`

Возвращает строку с текущим рабочим каталогом.

`getcwdu()`

Возвращает строку Юникода с текущим рабочим каталогом.

`getegid()`

Возвращает действующий идентификатор группы (UNIX).

`geteuid()`

Возвращает действующий идентификатор пользователя (UNIX).

`getgid()`

Возвращает действительный идентификатор группы для текущего процесса (UNIX).

`getgroups()`

Возвращает список целочисленных идентификаторов групп, которым принадлежит владелец процесса (UNIX).

`getlogin()`

Возвращает имя пользователя, ассоциированное с действующим идентификатором пользователя (UNIX).

`getpgid(pid)`

Возвращает идентификатор группы процессов для процесса с идентификатором *pid*. Если в аргументе *pid* передать значение 0, вернет идентификатор группы процессов для вызывающего процесса (UNIX).

`getpgrp()`

Возвращает идентификатор группы процессов для текущего процесса. Идентификаторы групп процессов обычно используются в механизмах управления заданиями. Идентификатор группы процессов – это не то же самое, что идентификатор группы процесса (UNIX).

`getpid()`

Возвращает действительный идентификатор текущего процесса (UNIX и Windows).

`getppid()`

Возвращает идентификатор родительского процесса (UNIX).

`getsid(pid)`

Возвращает идентификатор сеанса для процесса с идентификатором *pid*. Если в аргументе *pid* передать значение 0, вернет идентификатор сеанса для текущего процесса (UNIX).

`getuid()`

Возвращает действительный идентификатор пользователя для текущего процесса (UNIX).

`putenv(`*`varname`*`, `*`value`*`)`

Записывает значение *value* в переменную окружения *varname*. Изменения будут доступны дочерним процессам, запущенным с помощью функций `os.system()`, `popen()`, `fork()` и `execv()`. Операция присваивания значений элементам словаря `os.environ` автоматически вызывает функцию `putenv()`. Однако вызов `putenv()` не изменяет значения в `os.environ` (UNIX и Windows).

`setegid(`*`egid`*`)`

Устанавливает действующий идентификатор группы (UNIX).

`seteuid(euid)`

Устанавливает действующий идентификатор пользователя (UNIX).

`setgid(`*`gid`*`)`

Устанавливает идентификатор группы для текущего процесса (UNIX).

`setgroups(`*`groups`*`)`

Устанавливает групповые права доступа для текущего процесса. В аргументе *groups* передается последовательность целочисленных идентификаторов групп. Может вызываться только пользователем `root` (UNIX).

`setpgrp()`

Создает новую группу процессов с помощью системного вызова `setpgrp()` или `setpgrp(0, 0)`, в зависимости от того, какая версия реализована (если имеется). Возвращает идентификатор новой группы процессов (UNIX).

`setpgid(`*`pid`*`, `*`pgrp`*`)`

Присоединяет процесс с идентификатором *pid* к группе процессов *pgrp*. Если значение *pid* совпадает со значением *pgrp*, процесс становится лидером новой группы процессов. Если значение *pid* не совпадает со значением *pgrp*, процесс просто присоединяется к существующей группе процессов. Если в аргументе *pid* передать значение 0, будет использоваться идентификатор вызывающего процесса. Если в аргументе *pgrp* передать значение 0, процесс с идентификатором *pid* станет лидером группы процессов (UNIX).

`setreuid(`*`ruid`*`,`*`euid`*`)`

Устанавливает действительный и действующий идентификатор пользователя для вызывающего процесса (UNIX).

`setregid(`*`rgid`*`,`*`egid`*`)`

Устанавливает действительный и действующий идентификатор группы для вызывающего процесса (UNIX).

`setsid()`

Создает новый сеанс и возвращает его идентификатор. Сеансы обычно ассоциируются с устройствами терминалов и используются для управления процессами, запущенными в них (UNIX).

`setuid(uid)`

Устанавливает действительный идентификатор пользователя для текущего процесса. Эта функция требует наличия определенных привилегий и часто может использоваться только в процессах, запущенных пользователем `root` (UNIX).

`strerror(code)`

Возвращает текст сообщения об ошибке, соответствующий целочисленному коду *code* (UNIX и Windows). Символические имена кодов ошибок определены в модуле `errno`.

`umask(mask)`

Устанавливает маску `umask` для процесса и возвращает предыдущее значение `umask`. Маска `umask` используется для сброса битов разрешений у файлов, создаваемых процессом (UNIX и Windows).

`uname()`

Возвращает кортеж строк (*sysname*, *nodename*, *release*, *version*, *machine*), идентифицирующих тип операционной системы (UNIX).

`unsetenv(name)`

Сбрасывает значение переменной окружения *name*.

Создание файлов и дескрипторы файлов

Ниже перечислены функции, составляющие низкоуровневый интерфейс манипулирования файлами и каналами. Эти функции оперируют файлами, используя целочисленные дескрипторы *fd*. Дескриптор файла можно получить вызовом метода `fileno()` объекта файла.

`close(fd)`

Закрывает дескриптор файла *fd*, полученный ранее функцией `open()` или `pipe()`.

`closerange(low, high)`

Закрывает все дескрипторы файлов *fd* в диапазоне *low* <= *fd* < *high*. Возникающие ошибки игнорируются.

`dup(fd)`

Создает копию дескриптора *fd*. Возвращает новый дескриптор файла, который получает наименьшее возможное для процесса значение из числа дескрипторов. Старый и новый дескрипторы являются полностью эквивалентными друг другу. Кроме того, они совместно используют такие атрибуты файла, как указатель текущей позиции и блокировки (UNIX и Windows).

`dup2(oldfd, newfd)`

Копирует дескриптор *oldfd* в *newfd*. Если дескриптор *newfd* соответствует уже открытому файлу, он будет закрыт перед копированием (UNIX и Windows).

`fchmod(fd, mode)`

Устанавливает новый режим *mode* файла, соответствующего дескриптору *fd*. Дополнительная информация о режимах приводится в описании функции `os.open()` **(UNIX).**

`fchown(fd, uid, gid)`

Устанавливает новый идентификатор пользователя *uid* и группы *gid* владельца файла, соответствующего дескриптору *fd*. Чтобы сохранить один из идентификаторов неизменным, в соответствующем аргументе можно передать значение `-1` **(UNIX).**

`fdatasync(fd)`

Принудительно выталкивает на диск все кэшированные данные, записанные в файловый дескриптор *fd* **(UNIX).**

`fdopen(fd [, mode [, bufsize]])`

Создает открытый объект файла, подключенный к дескриптору *fd*. Аргументы *mode* и *bufsize* имеют тот же смысл, что и во встроенной функции `open()`. В аргументе *mode* должна передаваться строка, такая как `'r'`, `'w'` или `'a'`. В Python 3 этой функции могут передаваться любые дополнительные аргументы, которые принимает встроенная функция `open()`, такие как название кодировки или строка, определяющая символ окончания строк. Однако когда используется Python 2 и переносимость программы имеет большое значение, в вызовах этой функции следует использовать только аргументы *mode* и *bufsize*, описанные здесь.

`fpathconf(fd, name)`

Возвращает значения системных параметров настройки, имеющих отношение к файлам и каталогам, ассоциированных с открытым дескриптором файла *fd*. В аргументе *name* передается строка с именем требуемого параметра. Возвращаемые значения обычно извлекаются из определений параметров в системных заголовочных файлах, таких как `<limits.h>` и `<unistd.h>`. Стандарт **POSIX определяет следующие константы, которые могут передаваться** в аргументе *name*:

Константа	Описание
`"PC_ASYNC_IO"`	Указывает, могут ли выполняться асинхронные операции ввода-вывода над дескриптором *fd*.
`"PC_CHOWN_RESTRICTED"`	Указывает, может ли использоваться функция `chown()`. Если дескриптор *fd* ссылается на каталог, возвращаемый результат в равной степени относится ко всем файлам в этом каталоге.
`"PC_FILESIZEBITS"`	Максимально возможный размер файла.
`"PC_LINK_MAX"`	Максимальное значение счетчика ссылок для одного файла.
`"PC_MAX_CANON"`	Максимальная длина форматированной строки ввода. Дескриптор *fd* должен ссылаться на терминал.

(продолжение)

Константа	Описание
"PC_MAX_INPUT"	Максимальная длина строки ввода. Дескриптор *fd* должен ссылаться на терминал.
"PC_NAME_MAX"	Максимальная длина имени файла или каталога.
"PC_NO_TRUNC"	Указывает, должна ли приводить к ошибке ENAMETOOLONG попытка создать файл с именем, длина которого превышает значение PC_NAME_MAX.
"PC_PATH_MAX"	Максимальная длина относительного пути к файлу, когда каталог *fd* является текущим рабочим каталогом.
"PC_PIPE_BUF"	Размер буфера канала, когда дескриптор *fd* ссылается на канал или на очередь FIFO.
"PC_PRIO_IO"	Указывает, поддерживаются ли приоритетные операции ввода-вывода над дескриптором *fd*.
"PC_SYNC_IO"	Указывает, могут ли выполняться синхронные операции ввода-вывода над дескриптором *fd*.
"PC_VDISABLE"	Указывает, может ли быть запрещено действие специальных символов. Дескриптор *fd* должен ссылаться на терминал.

Перечисленные имена доступны не во всех платформах, а некоторые системы могут объявлять дополнительные параметры настройки. Перечень имен, известных в операционной системе, можно найти в словаре os.pathconf_names. Если параметр настройки известен, но его имя не включено в os.pathconf_names, в аргументе *name* можно передать его числовой идентификатор. Даже если имя параметра распознается интерпретатором, эта функция все равно может возбуждать исключение OSError, если сама операционная система не распознает параметр или этот параметр не связан с файлом *fd*. Эта функция доступна только в некоторых версиях UNIX.

`fstat(`*`fd`*`)`

Возвращает состояние дескриптора *fd*. Возвращает те же значения, что и функция os.stat() (UNIX и Windows).

`fstatvfs(`*`fd`*`)`

Возвращает информацию о файловой системе, где расположен файл, ассоциированный с дескриптором *fd*. Возвращает те же самые значения, что и функция os.statvfs() (UNIX).

`fsync(`*`fd`*`)`

Принудительно записывает на диск все незаписанные данные, имеющиеся в *fd*. Обратите внимание, что если вы пользуетесь объектом, поддерживающим буферизованные операции ввода-вывода (например, объектом file), то перед вызовом fsync() должна вызываться функция fdatasync(). Доступна в UNIX и Windows.

`ftruncate(fd, length)`

Усекает размер файла, соответствующего дескриптору *fd*, до значения *length* (UNIX).

`isatty(fd)`

Возвращает `True`, если дескриптор *fd* ссылается на TTY-подобное устройство, такое как терминал (UNIX).

`lseek(fd, pos, how)`

Устанавливает указатель текущей позиции для дескриптора *fd* в местоположение *pos*. Значение *how* интерпретируется следующим образом: `SEEK_SET` – позиционирование выполняется относительно начала файла, `SEEK_CUR` – позиционирование выполняется относительно текущей позиции и `SEEK_END` – позиционирование выполняется относительно конца файла. В старых программах на языке Python часто можно встретить использование вместо этих констант их числовых значений: `0`, `1` и `2` соответственно.

`open(file [, flags [, mode]])`

Открывает файл *file*. В аргументе *flags* передается битная маска, составленная из значений следующих констант, объединенных битовой операцией ИЛИ:

Значение	Описание
`O_RDONLY`	Открыть файл для чтения.
`O_WRONLY`	Открыть файл для записи.
`O_RDWR`	Открыть файл для чтения и записи (для изменения).
`O_APPEND`	Открыть файл для добавления в конец.
`O_CREAT`	Создать файл, если он не существует.
`O_NONBLOCK`	Установить неблокирующий режим выполнения операций создания, чтения и записи (UNIX).
`O_NDELAY`	То же, что и `O_NONBLOCK` (UNIX).
`O_DSYNC`	Установить режим синхронной записи (UNIX).
`O_NOCTTY`	При открытии устройства не назначать его управляющим терминалом (UNIX).
`O_TRUNC`	Если файл существует, усечь его длину до нуля.
`O_RSYNC`	Установить режим синхронного чтения (UNIX).
`O_SYNC`	Установить режим синхронной записи (UNIX).
`O_EXCL`	Сообщать об ошибке, если файл существует и установлен флаг `O_CREAT`.
`O_EXLOCK`	Установить исключительную блокировку на файл.
`O_SHLOCK`	Установить разделяемую блокировку на файл.
`O_ASYNC`	Установить асинхронный режим ввода, когда при появлении входных данных генерируется сигнал `SIGIO`.

(продолжение)

Значение	Описание
O_DIRECT	Установить режим непосредственного ввода-вывода, когда операции чтения и записи выполняются непосредственно с диском, а не с системными буферами.
O_DIRECTORY	Возбудить исключение, если файл не является каталогом.
O_NOFOLLOW	Не следовать за символическими ссылками.
O_NOATIME	Не обновлять время последнего обращения к файлу.
O_TEXT	Установить текстовый режим (Windows).
O_BINARY	Установить двоичный режим (Windows).
O_NOINHERIT	Файл не должен наследоваться дочерними процессами.
O_SHORT_LIVED	Подсказать системе, что файл используется для сохранения данных на короткое время (Windows).
O_TEMPORARY	Удалить файл после закрытия (Windows).
O_RANDOM	Подсказать системе, что файл используется в режиме произвольного доступа (Windows).
O_SEQUENTIAL	Подсказать системе, что файл используется в режиме последовательного доступа (Windows).

При использовании синхронных режимов (O_SYNC, O_DSYNC, O_RSYNC) операции ввода-вывода принудительно блокируются, пока они фактически не будут завершены на аппаратном уровне (например, операция записи вернет управление вызывающей программе, только когда байты физически будут записаны на диск). В аргументе *mode* передается битная маска с разрешениями, составленная из значений следующих восьмеричных значений (которые объявлены как константы в модуле stat), объединенных битовой операцией ИЛИ:

Режим	Описание
0100	Пользователь обладает правом на выполнение (stat.S_IXUSR).
0200	Пользователь обладает правом на запись (stat.S_IWUSR).
0400	Пользователь обладает правом на чтение (stat.S_IRUSR).
0700	Пользователь обладает правом на чтение/запись/выполнение (stat.S_IRWXU).
0010	Группа обладает правом на выполнение (stat.S_IXGRP).
0020	Группа обладает правом на запись (stat.S_IWGRP).
0040	Группа обладает правом на чтение (stat.S_IRGRP).
0070	Группа обладает правом на чтение/запись/выполнение (stat.S_IRWXG).
0001	Остальные обладают правом на выполнение (stat.S_IXOTH).

Режим	Описание
0002	Остальные обладают правом на запись (stat.S_IWOTH).
0004	Остальные обладают правом на чтение (stat.S_IROTH).
0007	Остальные обладают правом на чтение/запись/выполнение (stat.S_IRWXO).
4000	Установить режим UID (stat.S_ISUID).
2000	Установить режим GID (stat.S_ISGID).
1000	Установить бит закрепления (stat.S_ISVTX).

По умолчанию в аргументе *mode* передается значение 0777 & ~umask, где параметр umask используется для удаления выбранных разрешений. Например, значение 0022 в параметре umask приведет к удалению разрешения на запись для группы и остальных. Изменить значение umask можно с помощью функции os.umask(). Настройка параметра umask не оказывает никакого эффекта в Windows.

`openpty()`

Открывает псевдотерминал и возвращает пару дескрипторов (*master*, *slave*) для устройств PTY и TTY. Доступна в некоторых версиях UNIX.

`pipe()`

Создает неименованный канал, который может использоваться для однонаправленного взаимодействия с другим процессом. Возвращает пару дескрипторов (*r*, *w*), которые могут использоваться для чтения и записи соответственно. Обычно **эта функция вызывается непосредственно перед функцией** fork(). После вызова fork() передающий процесс закрывает канал, открытый для чтения, а принимающий процесс закрывает канал, открытый на запись. С этого момента канал активируется и может использоваться для передачи данных от одного процесса другому с помощью функций write() и read() (UNIX).

`read(`*fd*, *n*`)`

Читает до *n* байтов из дескриптора *fd*. Возвращает строку прочитанных байтов.

`tcgetpgrp(`*fd*`)`

Возвращает идентификатор группы процессов для управляющего терминала, заданного дескриптором *fd* (UNIX).

`tcsetpgrp(`*fd*, *pg*`)`

Устанавливает идентификатор группы процессов для управляющего терминала, заданного дескриптором *fd* (UNIX).

`ttyname(`*fd*`)`

Возвращает строку с именем файла устройства терминала, на который ссылается дескриптор *fd*. Если дескриптор *fd* не связан с устройством терминала, возбуждается исключение OSError (UNIX).

`write(fd, str)`

Записывает строку байтов *str* в файл с дескриптором *fd*. Возвращает количество байтов, которые фактически были записаны.

Файлы и каталоги

Следующие функции и переменные используются при работе с файлами и каталогами в файловой системе. Чтобы обойти различия в схемах именования файлов, можно использовать следующие переменные, содержащие информацию об устройстве имен в файловой системе:

Переменная	Описание
`altsep`	Альтернативный символ, используемый операционной системой для отделения элементов пути в файловой системе или `None`, если существует только один символ-разделитель. В DOS и Windows эта переменная имеет значение '/', где переменная `sep` содержит символ обратного слэша.
`curdir`	Строка, используемая для ссылки на текущий рабочий каталог: '.' – в UNIX и Windows, и ':' – в Macintosh.
`devnul`	Путь к пустому устройству (например, `/dev/null`).
`extsep`	Символ, отделяющий основное имя файла от расширения (например, символ '.' в имени '`foo.txt`').
`pardir`	Строка, используемая для ссылки на родительский каталог: '..' – в UNIX и Windows, и '::' – в Macintosh.
`pathsep`	Символ, используемый для отделения компонентов пути поиска (например, в переменной окружения `$PATH`): ':' – в UNIX, ';' – в DOS и Windows.
`sep`	Символ, используемый для отделения элементов пути в файловой системе: '/' – в UNIX и Windows, и ':' в Macintosh.

Для работы с файлами используются следующие функции:

`access(path, accessmode)`

Проверяет **разрешения для данного процесса на чтение/запись/выполнение** файла *path*. В аргументе *accessmode* передается значение `R_OK`, `W_OK`, `X_OK` или `F_OK` для проверки разрешения на чтение, запись, выполнение или проверки на существование соответственно. Возвращает `1`, если разрешение имеется, `0` – если нет.

`chflags(path, flags)`

Изменяет флаги для файла *path*. В аргументе *flags* передается битная маска, составленная из значений констант, перечисленных ниже, объединенных битовой операцией ИЛИ. Флаги, имена которых начинаются с `UF_`, могут устанавливаться любым пользователем, тогда как флаги, имена которых начинаются с `SF_`, могут изменяться только суперпользователем (UNIX).

Флаг	Описание
stat.UF_NODUMP	Не включать файл в вывод.
stat.UF_IMMUTABLE	Файл доступен только для чтения.
stat.UF_APPEND	Файл поддерживает только операцию добавления в конец.
stat.UF_OPAQUE	Каталог непрозрачен.
stat.UF_NOUNLINK	Файл не может быть удален или переименован.
stat.SF_ARCHIVED	Файл может быть архивирован.
stat.SF_IMMUTABLE	Файл доступен только для чтения.
stat.SF_APPEND	Файл поддерживает только операцию добавления в конец.
stat.SF_NOUNLINK	Файл не может быть удален или переименован.
stat.SF_SNAPSHOT	Файл является файлом мгновенного снимка.

`chmod(`*`path`*`, `*`mode`*`)`

Изменяет режим доступа к файлу *path*. Аргумент *mode* может принимать те же значения, что и в функции `open()`, описанной выше (UNIX и Windows).

`chown(`*`path`*`, `*`uid`*`, `*`gid`*`)`

Изменяет идентификатор владельца и группы для файла *path* в соответствии с числовыми значениями *uid* и *gid*. Чтобы сохранить один из идентификаторов неизменным, в соответствующем аргументе *uid* или *gid* можно передать значение `-1` (UNIX).

`lchflags(`*`path`*`, `*`flags`*`)`

То же, что и `chflags()`, но не следует по символическим ссылкам (UNIX).

`lchmod(`*`path`*`, `*`mode`*`)`

То же, что и `chflags()`, за исключением того, что если файл *path* является символической ссылкой, изменения будут затрагивать саму ссылку, а не файл, на который она ссылается.

`lchown(`*`path`*`, `*`uid`*`, `*`gid`*`)`

То же, что и `chown()`, но не следует по символическим ссылкам (UNIX).

`link(`*`src`*`, `*`dst`*`)`

Создает жесткую ссылку с именем *dst*, указывающую на файл *src* (UNIX).

`listdir(`*`path`*`)`

Возвращает список с именами элементов каталога *path*. Элементы могут следовать в списке в произвольном порядке, и в их число не включаются специальные элементы '.' и '..'. Если в аргументе *path* передается строка Юникода, возвращаемый список будет содержать только строки Юникода. Имейте в виду, что если имя какого-либо файла в каталоге не сможет быть преобразовано в Юникод, оно будет просто пропущено. Если в аргументе *path* передается строка байтов, то все имена в списке будут представлены строками байтов.

`lstat(`*`path`*`)`

Напоминает функцию `stat()`, но не следует по символическим ссылкам (UNIX).

`makedev(`*`major`*`, `*`minor`*`)`

Создает номер устройства, где аргументы *`major`* и *`minor`* определяют старший и младший номер устройства соответственно (UNIX).

`major(`*`devicenum`*`)`

Возвращает старший номер устройства для номера устройства *`devicenum`*, созданным вызовом функции `makedev()`.

`minor(`*`devicenum`*`)`

Возвращает младший номер устройства для номера устройства *`devicenum`*, созданным вызовом функции `makedev()`.

`makedirs(`*`path`*` [, `*`mode`*`])`

Функция рекурсивного создания каталога. Напоминает функцию `mkdir()`, но создает все промежуточные каталоги, содержащие конечный каталог. Если конечный каталог в пути *`path`* уже существует или не может быть создан, возбуждает исключение `OSError`.

`mkdir(`*`path`*` [, `*`mode`*`])`

Создает каталог *`path`* с флагами режима *`mode`*. По умолчанию аргумент *`mode`* принимает значение `0777`. В системах, отличных от UNIX, аргумент *`mode`* может не учитываться или игнорироваться.

`mkfifo(`*`path`*` [, `*`mode`*`])`

Создает очередь FIFO (именованный канал) с именем *`path`* и с флагами режима *`mode`*. По умолчанию аргумент *`mode`* принимает значение `0666` (UNIX).

`mknod(`*`path`*` [, `*`mode`*`, `*`device`*`])`

Создает файл устройства. В аргументе *`path`* передается имя файла, аргумент *`mode`* определяет права доступа и тип файла, а аргумент *`device`* – номер устройства, созданный с помощью функции `os.makedev()`. Аргумент *`mode`* может принимать те же значения, что и в функции `open()`, определяющие права доступа к файлу. Кроме того, для определения типа файла в аргумент *`mode`* добавляются флаги `stat.S_IFREG`, `stat.S_IFCHR`, `stat.S_IFBLK` и `stat.S_IFIFO` (UNIX).

`pathconf(`*`path`*`, `*`name`*`)`

Возвращает значения системных параметров настройки, имеющих отношение к имени *`path`*. В аргументе *`name`* передается строка с именем требуемого параметра. Имена доступных параметров были перечислены в описании функции `fpathconf()` (UNIX).

`readlink(`*`path`*`)`

Возвращает строку, содержащую путь, на который указывает символическая ссылка *`path`* (UNIX).

`remove(path)`

Удаляет файл *path*. Идентична функции `unlink()`.

`removedirs(path)`

Функция рекурсивного удаления каталогов. Действует так же, как функция `rmdir()`, за исключением того, что в случае успешного удаления конечного каталога она последовательно начнет удалять каталоги в пути *path*, начиная с самого правого, пока не будут удалены все указанные каталоги либо пока не будет встречена ошибка (которая игнорируется, потому что обычно появление ошибки означает, что родительский каталог не пуст). Если конечный каталог в пути *path* не может быть удален, возбуждает исключение `OSError`.

`rename(src, dst)`

Переименовывает файл или каталог *src* в *dst*.

`renames(old, new)`

Выполняет рекурсивное переименование каталогов или файлов. Действует так же, как функция `rename()`, за исключением того, что сначала пытается создать все промежуточные каталоги в пути *new*. После переименования она последовательно удаляет каталоги в пути *old*, начиная с самого правого, с помощью функции `removedirs()`.

`rmdir(path)`

Удаляет каталог *path*.

`stat(path)`

Выполняет системный вызов `stat()` для файла с именем *path*, чтобы получить информацию о нем. Возвращает объект, атрибуты которого содержат информацию о файле. Ниже перечислены наиболее типичные атрибуты:

Атрибут	Описание
`st_mode`	Режим доступа.
`st_ino`	Номер индексного узла (inode).
`st_dev`	Номер устройства, где располагается индексный узел.
`st_nlink`	Количество ссылок на индексный узел.
`st_uid`	Идентификатор пользователя, которому принадлежит файл.
`st_gid`	Идентификатор группы, которой принадлежит файл.
`st_size`	Размер файла в байтах.
`st_atime`	Время последнего обращения к файлу.
`st_mtime`	Время последнего изменения файла.
`st_ctime`	Время последнего изменения информации о состоянии файла.

В зависимости от типа операционной системы этот объект может содержать дополнительные атрибуты. Объект, возвращаемый функцией `stat()`,

может также рассматриваться как кортеж из 10 элементов (*st_mode*, *st_ino*, *st_dev*, *st_nlink*, *st_uid*, *st_gid*, *st_size*, *st_atime*, *st_mtime*, *st_ctime*). Последняя форма представления объекта предоставляется для сохранения обратной совместимости. Модуль `stat` определяет константы, которые могут использоваться для получения значений полей в этом кортеже.

`stat_float_times([`*newvalue*`])`

Возвращает `True`, если значения времени, возвращаемые функцией `stat()`, являются не целыми числами, а числами с плавающей точкой. Поведение функции можно изменить, передав логическое значение в аргументе *newvalue*.

`statvfs(`*path*`)`

Производит системный вызов `statvfs()` для указанного пути *path*, чтобы получить информацию о файловой системе. Возвращает объект, атрибуты которого содержат информацию о файловой системе. Ниже перечислены наиболее типичные атрибуты:

Атрибут	Описание
`f_bsize`	Рекомендуемый размер блока
`f_frsize`	Базовый размер блока
`f_blocks`	Общее количество блоков в файловой системе
`f_bfree`	Общее количество свободных блоков
`f_bavail`	Количество свободных блоков, доступных обычному пользователю
`f_files`	Общее количество индексных узлов
`f_ffree`	Общее количество свободных индексных узлов
`f_favail`	Количество свободных индексных узлов, доступных обычному пользователю
`f_flag`	Флаги (в зависимости от системы)
`f_namemax`	Максимальная длина имени файла

Возвращаемый объект может вести себя, как кортеж, содержащий значения этих атрибутов в указанном порядке. Стандартный модуль `statvfs` объявляет константы, которые могут использоваться для извлечения информации из этого кортежа (UNIX).

`symlink(`*src*`, `*dst*`)`

Создает символическую ссылку с именем *dst*, которая указывает на *src*.

`unlink(`*path*`)`

Удаляет файл *path*. То же, что и функция `remove()`.

`utime(`*path*`, (`*atime*`, `*mtime*`))`

Устанавливает время последнего обращения и время последнего изменения файла в соответствии с указанными значениями. (Во втором аргумен-

те передается кортеж из двух элементов.) Значения времени указываются в том же виде, в каком они возвращаются функцией time.time().

walk(*top* [, *topdown* [, *onerror* [,*followlinks*]]])

Создает объект-генератор, который выполняет обход дерева каталогов. Аргумент *top* определяет самый верхний каталог. В аргументе *topdown* передается логическое значение, определяющее направление обхода дерева каталогов – сверху вниз (по умолчанию) или снизу вверх. Возвращаемый генератор воспроизводит кортежи вида (*dirpath*, *dirnames*, *filenames*), где *dirpath* – строка, содержащая путь к каталогу, *dirnames* – список всех подкаталогов в каталоге *dirpath* и *filenames* – список файлов в каталоге *dirpath*, за исключением каталогов. В аргументе *onerror* **передается функция, принимающая** единственный аргумент. Эта функция будет вызываться при появлении любых ошибок в процессе обхода, и ей будет передаваться экземпляр os.error. По умолчанию все ошибки игнорируются. Если обход каталога выполняется в направлении сверху вниз, изменения в списке каталогов *dirnames* будут оказывать влияние на процесс обхода. Например, если из списка *dirnames* удалить какие-либо каталоги, они будут пропущены. По умолчанию генератор не следует по символическим ссылкам, если в аргументе *followlinks* не передать значение True.

Управление процессами

Следующие функции и переменные используются для создания, уничтожения и управления процессами:

abort()

Генерирует сигнал SIGABRT, который отправляется вызывающему процессу. Если сигнал не обрабатывается процессом, по умолчанию это приводит к завершению процесса с сообщением об ошибке.

defpath

Эта переменная содержит путь поиска по умолчанию, используемый функциями exec*p*(), когда окружение не содержит переменную 'PATH'.

execl(*path*, *arg0*, *arg1*, ...)

Эквивалентна вызову функции execv(*path*, (*arg0*, *arg1*, ...)).

execle(*path*, *arg0*, *arg1*, ..., *env*)

Эквивалентна вызову функции execve(*path*, (*arg0*, *arg1*, ...), *env*).

execlp(*path*, *arg0*, *arg1*, ...)

Эквивалентна вызову функции execvp(*path*, (*arg0*, *arg1*, ...)).

execv(*path*, *args*)

Запускает программу *path* со списком аргументов *args*, замещая текущей процесс (то есть интерпретатор Python). Список аргументов может быть представлен кортежем или списком строк.

execve(*path*, *args*, *env*)

Запускает новую программу, как это делает функция execv(), но дополнительно принимает словарь *env*, определяющий окружение, в котором будет

выполняться программа. В аргументе *env* должен передаваться словарь, отображающий строки в строки.

`execvp(path, args)`

Похожа на функцию `execv(path, args)`, но при этом повторяет поведение командной оболочки, отыскивая выполняемый файл в списке каталогов. Список каталогов извлекается из словаря `environ['PATH']`.

`execvpe(path, args, env)`

Похожа на функцию `execvp()`, но дополнительно принимает словарь *env*, определяющий окружение процесса, как это делает функция `execve()`.

`_exit(n)`

Немедленно завершает процесс с кодом завершения *n*, без выполнения заключительных операций. Обычно эта функция вызывается только в дочерних процессах, созданных вызовом функции `fork()`. Вызов этой функции отличается от вызова `sys.exit()`, которая выполняет все необходимые заключительные операции. Значение кода завершения зависит от приложения, но обычно значение 0 свидетельствует об успешном завершении, тогда как ненулевое значение указывает на то, что произошла какая-то ошибка. В зависимости от типа операционной системы могут быть определены следующие стандартные коды завершения:

Значение	Описание
EX_OK	Нет ошибки.
EX_USAGE	Неверная команда.
EX_DATAERR	Недопустимые входные данные.
EX_NOINPUT	Отсутствуют входные данные.
EX_NOUSER	Пользователь не существует.
EX_NOHOST	Хост не существует.
EX_NOTFOUND	Не найдено.
EX_UNAVAILABLE	Служба недоступна.
EX_SOFTWARE	Внутренняя ошибка программы.
EX_OSERR	Ошибка операционной системы.
EX_OSFILE	Ошибка файловой системы.
EX_CANTCREAT	Невозможно создать выходной файл.
EX_IOERR	Ошибка ввода-вывода.
EX_TEMPFAIL	Временная ошибка.
EX_PROTOCOL	Ошибка протокола.
EX_NOPERM	Недостаточно прав.
EX_CONFIG	Ошибка настройки.

`fork()`

Создает дочерний процесс. Возвращает 0 во вновь созданном дочернем процессе и идентификатор дочернего процесса – в оригинальном процессе. Дочерний процесс является точной копией оригинального процесса, и ему доступны многие ресурсы родительского процесса, такие как открытые файлы (UNIX).

`forkpty()`

Создает дочерний процесс, использующий новый псевдотерминал в качестве управляющего терминала. Возвращает пару значений (*pid*, *fd*), где *pid* имеет значение 0 в дочернем процессе, а *fd* содержит дескриптор псевдотерминала с ведущей стороны. Эта функция доступна только в некоторых версиях UNIX.

`kill(`*pid*, *sig*`)`

Посылает сигнал *sig* процессу *pid*. Перечень имен сигналов можно найти в модуле `signal` (UNIX).

`killpg(`*pgid*, *sig*`)`

Посылает сигнал *sig* группе процессов *pgid*. Перечень имен сигналов можно найти в модуле `signal` (UNIX).

`nice(`*increment*`)`

Добавляет значение *increment* к приоритету процесса. Возвращает новое значение приоритета. Обычно рядовые пользователи могут только понижать приоритет процесса, так как для повышения приоритета требуются привилегии пользователя `root`. Эффект от изменения приоритета зависит от типа операционной системы, но обычно понижение приоритета применяется к фоновым процессам, чтобы они не оказывали существенного влияния на производительность других процессов (UNIX).

`plock(`*op*`)`

Закрепляет сегменты программы в памяти, предотвращая их от вытеснения в файл подкачки. В аргументе *op* передается целое число, определяющее, какие сегменты должны быть закреплены. Допустимые значения аргумента *op* зависят от системы, но обычно в их число входят `UNLOCK`, `PROCLOCK`, `TXTLOCK` и `DATLOCK`. Эти константы не определены в языке Python, но их можно найти в заголовочном файле `<sys/lock.h>`. Данная функция доступна не во всех платформах и зачастую может вызываться только процессом с действующим идентификатором пользователя 0 (`root`) (UNIX).

`popen(`*command* `[,` *mode* `[,` *bufsize*`]])`

Открывает канал к команде *command* или от нее. Возвращает открытый объект файла, подключенный к каналу, доступный для чтения или для записи, в зависимости от значения аргумента *mode* – `'r'` (по умолчанию) или `'w'`. Аргумент *bufsize* имеет тот же смысл, что и во встроенной функции `open()`. Код завершения команды *command* возвращается методом `close()` возвращаемого объекта файла за исключением случая, когда код завершения равен нулю, – в этом случае возвращается `None`.

`spawnv(`*`mode`*`, `*`path`*`, `*`args`*`)`

Запускает программу *path* в новом процессе и передает ей аргументы *args* в виде параметров командной строки. В аргументе *args* допускается передавать список или кортеж. Первый элемент в *args* должен быть именем файла программы. В аргументе *mode* передается одна из следующих констант:

Константа	Описание
P_WAIT	Запускает программу и ожидает ее завершения. Возвращает код завершения программы.
P_NOWAIT	Запускает программу и возвращает дескриптор процесса.
P_NOWAITO	То же, что и P_NOWAIT.
P_OVERLAY	Запускает программу и уничтожает вызывающий процесс (действует как функция exec).
P_DETACH	Запускает программу и отсоединяется от нее. Вызывающая программа продолжает работу, но она уже не может контролировать порожденный процесс.

Функция spawnv() доступна в Windows и в некоторых версиях UNIX.

`spawnve(`*`mode`*`, `*`path`*`, `*`args`*`, `*`env`*`)`

Запускает программу *path* в новом процессе, передает ей аргументы *args* в виде параметров командной строки и содержимое отображения *env* в виде окружения. В аргументе *args* допускается передавать список или кортеж. Аргумент *mode* имеет тот же смысл, что и в функции spawnv().

`spawnl(`*`mode`*`, `*`path`*`, `*`arg1`*`, ... , `*`argn`*`)`

То же, что и spawnv(), за исключением того, что все параметры передаются в виде отдельных аргументов.

`spawnle(`*`mode`*`, `*`path`*`, `*`arg1`*`, ... , `*`argn`*`, `*`env`*`)`

То же, что и spawnve(), за исключением того, что все параметры передаются в виде отдельных аргументов. Последним аргументом предается отображение с переменными окружения.

`spawnlp(`*`mode`*`, `*`file`*`, `*`arg1`*`, ... , `*`argn`*`)`

То же, что и spawnl(), но поиск выполняемого файла *file* выполняется с использованием переменной окружения PATH (UNIX).

`spawnlpe(`*`mode`*`, `*`file`*`, `*`arg1`*`, ... , `*`argn`*`, `*`env`*`)`

То же, что и spawnle(), но поиск выполняемого файла *file* выполняется с использованием переменной окружения PATH (UNIX).

`spawnvp(`*`mode`*`, `*`file`*`, `*`args`*`)`

То же, что и spawnv(), но поиск выполняемого файла *file* выполняется с использованием переменной окружения PATH (UNIX).

`spawnvpe(`*`mode`*`, `*`file`*`, `*`args`*`, `*`env`*`)`

То же, что и spawnve(), но поиск выполняемого файла *file* выполняется с использованием переменной окружения PATH (UNIX).

`startfile(`*`path`* `[, `*`operation`*`])`

Запускает приложение, определяемое аргументом *path*. Выполняет те же действия, которые обычно выполняются в результате двойного щелчка мышью на файле в Windows Explorer. Функция возвращает управление, как только приложение будет запущено. Кроме того, она не обеспечивает возможности дождаться завершения приложения или получить код его завершения. Путь в аргументе *path* определяет местоположение выполняемого файла относительно текущего каталога. В аргументе *operation* можно передать строку, определяющую дополнительные действия, которые должны быть выполнены при открытии файла *path*. По умолчанию этот аргумент имеет значение `'open'`, но может быть установлен в значение `'print'`, `'edit'`, `'explore'` или `'find'` (зависит от типа файла (Windows)).

`system(`*`command`*`)`

Выполняет команду *command* (строка) в подоболочке. В UNIX возвращает код завершения процесса, который получает от функции `wait()`. В Windows всегда возвращает код завершения 0. Модуль `subprocess` предоставляет более мощный и более предпочтительный способ запуска дочерних процессов.

`times()`

Возвращает кортеж из 5 чисел с плавающей точкой, представляющих накопленное время в секундах. В UNIX кортеж содержит время выполнения в пространстве пользователя, время выполнения в пространстве ядра, время выполнения дочернего процесса в пространстве пользователя, время выполнения дочернего процесса в пространстве ядра и действительное время работы – именно в таком порядке. В Windows кортеж содержит время выполнения в пространстве пользователя, время выполнения в пространстве ядра и нули в трех остальных элементах.

`wait([`*`pid`*`])`

Ожидает завершения дочернего процесса и возвращает кортеж с идентификатором процесса и кодом завершения. Код завершения – это 16-битное число, младший байт которого определяет номер сигнала, вызвавшего завершение процесса, а старший байт – код завершения (если номер сигнала равен нулю). Если был создан core-файл с образом памяти процесса, устанавливается старший бит младшего байта. Аргумент *pid*, если указан, определяет процесс, завершения которого следует дождаться. При вызове без аргументов функция `wait()` возвращает управление, когда завершится любой из дочерних процессов (UNIX).

`waitpid(`*`pid`*`, `*`options`*`)`

Ожидает изменения состояния дочернего процесса с идентификатором *pid* и возвращает кортеж с идентификатором процесса и кодом завершения, который кодируется так же, как и в функции `wait()`. В нормальной ситуации в аргументе *options* следует передавать 0 или `WNOHANG`, чтобы избежать приостановки вызывающего процесса, если состояние дочернего процесса окажется недоступным немедленно. Эта функция может также использоваться для сбора информации о дочерних процессах, выполнение которых было всего лишь приостановлено по каким-либо причинам. Если в аргу-

менте *options* передать значение WCONTINUED, можно получить информацию о дочернем процессе, выполнение которого было возобновлено после приостановки механизмом управления заданиями. Если в аргументе *options* передать значение WUNTRACED, можно получить информацию о дочернем процессе, который был приостановлен, но чей код состояния еще недоступен.

wait3([*options*])

То же, что и wait() (без аргументов), за исключением того, что возвращает кортеж из 3 элементов (*pid, status, rusage*), где в поле *pid* возвращается идентификатор дочернего процесса, в поле *status* – код завершения, и в поле *rusage* – информация об использованных ресурсах, возвращаемая функцией resource.getrusage(). Аргумент *options* имеет тот же смысл, что и в функции waitpid().

wait4(*pid, options*)

То же, что и waitpid(), за исключением того, что возвращает тот же результат, что и функция wait3().

Следующие функции принимают код завершения процесса, возвращаемый функциями waitpid(), wait3() или wait4(), и используются для проверки состояния процесса (UNIX).

WCOREDUMP(*status*)

Возвращает True, если был создан core-файл с образом памяти процесса.

WIFEXITED(*status*)

Возвращает True, если процесс завершил работу обращением к системному вызову exit().

WEXITSTATUS(*status*)

Если функция WIFEXITED() возвращает True, эта функция вернет целочисленное значение аргумента системного вызова exit(). В противном случае вернет значение, не имеющее определенного смысла.

WIFCONTINUED(*status*)

Возвращает True, если процесс возобновил работу после приостановки механизмом управления заданиями.

WIFSIGNALED(*status*)

Возвращает True, если процесс завершил работу в результате получения сигнала.

WIFSTOPPED(*status*)

Возвращает True, если процесс был приостановлен.

WSTOPSIG(*status*)

Возвращает сигнал, вызвавший остановку процесса.

WTERMSIG(*status*)

Возвращает сигнал, вызвавший завершение процесса.

Параметры системы

Ниже перечислены функции, с помощью которых можно получить информацию о параметрах системы:

`confstr(`*`name`*`)`

Возвращает строковое значение параметра настройки системы. В аргументе *name* передается строка с именем параметра. Перечень допустимых имен зависит от системы, но его можно найти в `os.confstr_names`. Если значение параметра с указанным именем не определено, возвращается пустая строка. Если имя параметра неизвестно, возбуждается исключение `ValueError`. В случаях, когда система не поддерживает параметр с указанным именем, может также возбуждаться исключение `OSError`. Параметры, возвращаемые этой функцией, главным образом имеют отношение к окружению сборки программ и включают пути к системным утилитам, параметры компилятора для различных конфигураций (например, поддержка 32-битных приложений, 64-битных приложений и файлов большого размера) и параметры компоновщика (UNIX).

`getloadavg()`

Возвращает кортеж из 3 элементов со средними значениями занятости процессора за последние 1, 5 и 15 минут (UNIX).

`sysconf(`*`name`*`)`

Возвращает целочисленное значение системного параметра. В аргументе *name* передается строка с именем параметра. Перечень допустимых имен параметров системы можно найти в словаре `os.sysconf_names`. Возвращает -1, если имя параметра известно, но его значение не определено. В противном случае, если имя параметра неизвестно, может возбуждать исключение `ValueError` или `OSError`. В некоторых системах определено свыше 100 различных **системных параметров. В списке ниже перечислены параметры**, определяемые стандартом POSIX.1, которые должны быть доступны в большинстве версий UNIX:

Параметр	Описание
"SC_ARG_MAX"	Максимальная длина аргументов функции `exec()`.
"SC_CHILD_MAX"	Максимальное количество процессов на один идентификатор пользователя.
"SC_CLK_TCK"	Количество тактов системных часов в одной секунде.
"SC_NGROUPS_MAX"	Максимальное количество идентификаторов дополнительных групп на процесс.
"SC_STREAM_MAX"	Максимальное количество одновременно открытых потоков ввода-вывода на процесс.
"SC_TZNAME_MAX"	Максимальное количество байтов в названии часового пояса.
"SC_OPEN_MAX"	Максимальное количество одновременно открытых файлов на процесс.

(продолжение)

Параметр	Описание
"SC_JOB_CONTROL"	Признак поддержки системой механизма управления заданиями.
"SC_SAVED_IDS"	Признак поддержки системой сохраненных идентификаторов пользователя и группы для каждого процесса.

`urandom(`*n*`)`

Возвращает строку *n* байтов со случайными значениями, сгенерированными системой (например, с помощью устройства `/dev/urandom` в UNIX). Возвращаемые байты пригодны для использования в криптографических целях.

Исключения

Для уведомления об ошибках в модуле `os` объявляется единственное исключение.

`error`

Возбуждается, когда функция возвращает признак системной ошибки. Аналогично встроенному исключению `OSError`. Несет в себе два значения: `errno` и `strerr`. Первое значение представляет целочисленный код ошибки, из числа тех, что приводятся в описании модуля `errno`. Второе – строка с текстом сообщения об ошибке. Исключения, имеющие отношение к ошибкам, возникшим при выполнении операций с файловой системой, имеют третий дополнительный атрибут `filename` с именем файла, переданным функции.

Модуль os.path

Модуль `os.path` позволяет обеспечить переносимость операций со строками путей в файловой системе. Он импортируется модулем `os`.

`abspath(`*path*`)`

Возвращает абсолютную версию пути *path*, с учетом текущего рабочего каталога. Например, вызов `abspath('../Python/foo')` мог бы вернуть `'/home/beazley/Python/foo'`.

`basename(`*path*`)`

Возвращает базовое имя в пути *path*. Например, вызов `basename('/usr/local/python')` вернет `'python'`.

`commonprefix(`*list*`)`

Возвращает наибольшую строку, которая является префиксом для всех строк в списке *list*. Если список *list* пуст, возвращает пустую строку.

`dirname(`*path*`)`

Возвращает имя каталога для базового имени в пути *path*. Например, вызов `dirname('/usr/local/python')` вернет `'/usr/local'`.

`exists(path)`

Возвращает `True`, если путь *path* существует в файловой системе. Возвращает `False`, если указанный путь *path* не существует.

`expanduser(path)`

Замещает в пути ссылки вида '`~user`' полными путями к домашним каталогам пользователей. Если такая подстановка невозможна или строка *path* не начинается с символа '`~`', возвращает значение аргумента *path* без изменений.

`expandvars(path)`

Замещает в пути *path* ссылки на переменные вида '`$name`' или '`${name}`' их значениями. Если имя переменной указано неправильно или эта переменная не существует, возвращает значение аргумента *path* без изменений.

`getatime(path)`

Возвращает время последнего обращения в виде числа секунд, прошедших с начала эпохи (смотрите описание модуля `time`). Если функция `os.stat_float_times()` возвращает `True`, возвращаемое значение может быть числом с плавающей точкой.

`getctime(path)`

В UNIX возвращает время последнего изменения, а в Windows – время создания. Время возвращается в виде количества секунд, прошедших с начала эпохи (смотрите описание модуля `time`). В некоторых случаях возвращаемое значение может быть числом с плавающей точкой (смотрите описание функции `getatime()`).

`getmtime(path)`

Возвращает время последнего изменения в виде числа секунд, прошедших с начала эпохи (смотрите описание модуля `time`). В некоторых случаях возвращаемое значение может быть числом с плавающей точкой (смотрите описание функции `getatime()`).

`getsize(path)`

Возвращает размер файла в байтах.

`isabs(path)`

Возвращает `True`, если путь *path* является абсолютным путем (начинается с символа слэша).

`isfile(path)`

Возвращает `True`, если аргумент *path* содержит путь к обычному файлу. Эта функция разыменовывает символические ссылки, поэтому для одного и того же значения *path* обе функции, `islink()` и `isfile()`, могут возвращать `True`.

`isdir(path)`

Возвращает `True`, если аргумент *path* содержит путь к каталогу. Символические ссылки разыменовываются.

`islink(path)`

Возвращает `True`, если аргумент *path* содержит путь к символической ссылке. Если символические ссылки не поддерживаются системой, возвращает `False`.

`ismount(path)`

Возвращает `True`, если аргумент *path* содержит путь к точке монтирования.

`join(path1 [, path2 [, ...]])`

Составляет путь из одного или более аргументов. Например, вызов `join('home', 'beazley', 'Python')` вернет `'home/beazley/Python'`.

`lexists(path)`

Возвращает `True`, если путь *path* существует. Для всех символических ссылок возвращает `True`, даже если символические ссылки указывают на несуществующий файл или каталог.

`normcase(path)`

Нормализует регистр символов в пути *path*. Если файловая система нечувствительна к регистру символов, эта функция приведет все символы к нижнему регистру. Кроме того, в Windows заменит символы слэша символами обратного слэша.

`normpath(path)`

Нормализует путь *path*. Устраняет излишние разделители и ссылки на родительские каталоги. Например, все следующие пути: `'A//B'`, `'A/./B'` и `'A/foo/../B'` будут преобразованы в путь `'A/B'`. В Windows все символы слэша будут замещены символами обратного слэша.

`realpath(path)`

Возвращает действительный путь для пути *path*, устраняя все символические ссылки, если таковые имеются (UNIX).

`relpath(path [, start])`

Для пути *path* вернет относительный путь от текущего рабочего каталога. Аргумент *start* может использоваться, чтобы указать другой начальный каталог.

`samefile(path1, path2)`

Возвращает `True`, если *path1* и *path2* ссылаются на один и тот же файл или каталог (UNIX).

`sameopenfile(fp1, fp2)`

Возвращает `True`, если открытые объекты файлов *fp1* и *fp2* ссылаются на один и тот же файл (UNIX).

`samestat(stat1, stat2)`

Возвращает `True`, если кортежи *stat1* и *stat2*, полученные в результате вызова функции `fstat()`, `lstat()` или `stat()`, относятся к одному и тому же файлу (UNIX).

`split(path)`

Разбивает путь *path* на две части (*head*, *tail*), где поле *tail* содержит последний элемент пути, а поле *head* – все, что предшествует последнему элементу. Например, путь '/home/user/foo' будет разбит на части ('/home/ user', 'foo'). Тот же кортеж можно получить с помощью конструкции (dirname(), basename()).

`splitdrive(path)`

Разбивает путь *path* на две части (*drive*, *filename*), где поле *drive* содержит определение устройства или пустую строку. В системах, где отсутствует понятие «буква устройства», в поле *drive* всегда возвращается пустая строка.

`splitext(path)`

Разбивает путь *path* на базовое имя файла и расширение. Например, вызов `splitext('foo.txt')` вернет `('foo', '.txt')`.

`splitunc(path)`

Разбивает путь *path* на две части (unc,rest), где поле *unc* содержит строку с точкой монтирования в формате UNC (Universal Naming Convention – универсальное соглашение об именовании), а поле *rest* – остаток пути (Windows).

`supports_unicode_filenames`

Переменная, имеет значение `True`, если файловая система поддерживает символы Юникода в именах файлов.

Примечание

В системе Windows требуется проявлять особую осторожность при работе с именами файлов, включающими букву устройства (например, 'C:spam.txt'). В большинстве случаев предполагается, что файл находится в текущем рабочем каталоге. Например, если текущим рабочим каталогом является 'C:\Foo\', то строка с именем файла 'C:spam.txt' будет интерпретироваться, как путь 'C:\Foo\C:spam.txt', а не как 'C:\spam.txt'.

См. также

Описание модулей fnmatch (стр. 398), glob (стр. 399), os (стр. 475).

Модуль signal

Модуль `signal` используется для создания обработчиков сигналов в программах на языке Python. Сигналы, как правило, соответствуют асинхронным событиям и посылаются программе по истечении установленного интервала времени, при получении входных данных или в результате каких-либо действий пользователя. Интерфейс модуля `signal` имитирует интерфейс, реализованный в UNIX, несмотря на то, что отдельные части модуля могут поддерживать и другие платформы.

`alarm(time)`

Если аргумент *time* не равен нулю, запрашивает посылку сигнала SIGALRM текущему процессу через *time* секунд. Ранее запланированная посылка этого сигнала при этом отменяется. Если в аргументе *time* передать ноль, ранее запланированная посылка этого сигнала отменяется. Возвращает время в секундах, оставшееся до посылки ранее запланированного сигнала, или ноль, если посылка сигнала не была запланирована (UNIX).

`getsignal(signalnum)`

Возвращает обработчик сигнала *signalnum*. Возвращаемый объект является объектом Python, поддерживающим возможность вызова. Функция может также возвращать значение SIG_IGN, если указанный сигнал игнорируется, SIG_DFL, если сигнал обрабатывается обработчиком по умолчанию, или None, если обработчик сигнала не был установлен интерпретатором Python.

`getitimer(which)`

Возвращает текущее значение внутреннего таймера с идентификатором *which*.

`pause()`

Приостанавливает выполнение процесса до получения следующего сигнала (UNIX).

`set_wakeup_fd(fd)`

Определяет дескриптор файла, в который будет записан байт '\0' при получении сигнала. Это может использоваться для обработки сигналов в программах, которые циклически опрашивают дескрипторы файлов с помощью функций, например тех, что объявлены в модуле select. Файл, представленный дескриптором fd, должен быть открыт в неблокирующем режиме.

`setitimer(which, seconds [, interval])`

Устанавливает внутренний таймер, посылающий сигнал через *seconds* секунд и затем повторяющий его через каждые *interval* секунд. В обоих этих аргументах передаются числа с плавающей точкой. В аргументе *which* можно передать одно из следующих значений: ITIMER_REAL, ITIMER_VIRTUAL или ITIMER_PROF. Этот аргумент определяет, какой сигнал будет посылаться процессу по истечении таймера. Значение ITIMER_REAL соответствует сигналу SIGALRM, значение ITIMER_VIRTUAL – сигналу SIGVTALRM и значение ITIMER_PROF – сигналу SIGPROF. Значение 0 в аргументе *seconds* сбрасывает таймер. Возвращает кортеж (*seconds, interval*) с предыдущими параметрами таймера.

`siginterrupt(signalnum, flag)`

Определяет поведение системных вызовов при получении процессом сигнала *signalnum*. Если в аргументе *flag* передать False, при получении сигнала *signalnum* системный вызов будет автоматически перезапущен. Если передать True, работа системного вызова будет прервана. Как правило, прерванные системные вызовы возбуждают исключение OSError или IOError с кодом ошибки errno.EINTR или errno.EAGAIN.

`signal(`*`signalnum`*`, `*`handler`*`)`

Устанавливает функцию *handler* в качестве обработчика сигнала *signalnum*. В аргументе *handler* должен передаваться объект Python, поддерживающий возможность вызова и принимающий два аргумента: номер сигнала и объект кадра стека. Чтобы игнорировать сигнал или установить обработчик по умолчанию, можно передать значение `SIG_IGN` или `SIG_DFL` соответственно. Возвращает предыдущий обработчик сигнала, `SIG_IGN` или `SIG_DFL`. В многопоточных приложениях эта функция должна вызываться только из главного потока управления. В противном случае будет возбуждено исключение `ValueError`.

Отдельные сигналы могут идентифицироваться с помощью символических констант вида `SIG*`. Эти имена соответствуют целочисленным значениям, зависящим от типа системы. Ниже приводятся наиболее часто используемые константы:

Имя сигнала	Описание
SIGABRT	Аварийное прерывание процесса
SIGALRM	Генерируется таймером
SIGBUS	Аппаратная ошибка, обычно связанная с памятью
SIGCHLD	Изменилось состояние дочернего процесса
SIGCLD	Изменилось состояние дочернего процесса
SIGCONT	Продолжить работу приостановленного процесса
SIGFPE	Ошибка при выполнении операции над числами с плавающей точкой
SIGHUP	Разрыв связи с управляющим терминалом
SIGILL	Недопустимая инструкция
SIGINT	От терминала получен символ прерывания работы процесса
SIGIO	Событие асинхронного ввода-вывода
SIGIOT	Аппаратная ошибка
SIGKILL	Завершить работу процесса
SIGPIPE	Выполнена запись в канал, когда с другого конца канала нет читающего процесса
SIGPOLL	Произошло событие в опрашиваемом устройстве
SIGPROF	Генерируется профилирующим таймером
SIGPWR	Сбой в питании
SIGQUIT	От терминала получен символ завершения работы процесса
SIGSEGV	Обращение к недопустимому адресу в памяти
SIGSTOP	Останов процесса

(продолжение)

Имя сигнала	Описание
SIGTERM	Завершить работу процесса
SIGTRAP	Аппаратная ошибка
SIGTSTP	От терминала получен символ приостановки процесса
SIGTTIN	Управляющий сигнал от терминала
SIGTTOU	Управляющий сигнал от терминала
SIGURG	Произошло срочное событие
SIGUSR1	Определяется пользователем
SIGUSR2	Определяется пользователем
SIGVTALRM	Генерируется виртуальным таймером
SIGWINCH	Изменился размера окна
SIGXCPU	Достигнут предел на использование процессора
SIGXFSZ	Превышен предельный размер файла

Кроме того, в модуле объявлены следующие переменные:

Переменная	Описание
SIG_DFL	Обработчик сигнала, который вызывает обработчик по умолчанию
SIG_IGN	Обработчик сигнала, который игнорирует сигнал
NSIG	Имеет значение на единицу больше самого большого номера сигнала

Пример

Следующий пример демонстрирует, как ограничить время ожидания установления соединения (модуль `socket` уже поддерживает возможность ограничения по времени, поэтому данный пример предназначен лишь для того, чтобы продемонстрировать основные приемы работы с модулем `signal`).

```
import signal, socket
def handler(signum, frame):
    print 'Timeout!'
    raise IOError, 'Host not responding.'
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
signal.signal(signal.SIGALRM, handler)
signal.alarm(5)                    # Послать сигнал через 5 секунд
sock.connect('www.python.org', 80)  # Установить соединение
signal.alarm(0)                    # Сбросить таймер
```

Примечания

- Обработчики сигналов продолжают действовать, пока явно не будут сброшены. Исключение составляет сигнал SIGCHLD (поведение которого определяется реализацией).
- Не существует способа временно заблокировать сигналы.
- Обработчики сигналов вызываются асинхронно, но это может произойти только между атомарными инструкциями интерпретатора Python. Доставка сигнала может быть отложена на время, пока производятся длительные вычисления, реализованные на языке C (например, в модуле расширения).
- Если сигнал поступает во время выполнения операции ввода-вывода, она может быть прервана исключением. В этом случае в атрибут errno исключения записывается код ошибки errno.EINTR, чтобы показать, что работа системного вызова была прервана сигналом.
- Некоторые сигналы, такие как SIGSEGV, не могут обрабатываться из программного кода на языке Python.
- Интерпретатор Python устанавливает некоторое количество обработчиков сигналов по умолчанию. Сигнал SIGPIPE игнорируется, сигнал SIGINT преобразуется в исключение KeyboardInterrupt, а при получении сигнала SIGTERM производятся заключительные операции по завершению процесса и вызывается функция sys.exitfunc.
- Особую осторожность следует проявлять при использовании сигналов в многопоточных программах. В настоящее время установка новых обработчиков и прием сигналов может производиться только из главного потока.
- Возможности обработки сигналов в Windows достаточно ограничены. Количество сигналов, поддерживаемых в этой платформе, весьма невелико.

Модуль subprocess

Модуль subprocess содержит функции и классы, обеспечивающие универсальный интерфейс для выполнения таких задач, как создание новых процессов, управление потоками ввода и вывода и обработка кодов возврата. Модуль объединяет в себе многие функциональные возможности, присутствующие в других модулях, таких как os, popen2 и commands.

`Popen(`*`args`*`, **`*`parms`*`)`

Выполняет команду, запуская новый дочерний процесс, и возвращает объект класса Popen, представляющий новый процесс. Команда определяется в аргументе *args* либо как строка, такая как 'ls -l', либо как список строк, такой как ['ls', '-l']. В аргументе *parms* передается коллекция именованных аргументов, которые могут использоваться для управления различными характеристиками дочернего процесса. Ниже приводится перечень допустимых именованных аргументов:

Именованный аргумент	Описание
bufsize	Определяет режим буферизации, где значение 0 соответствует отсутствию буферизации, 1 – выполняется построчная буферизация, при отрицательном значении используются системные настройки, а любое другое положительное значение определяет примерный размер буфера. По умолчанию используется значение 0.
close_fds	Если имеет значение True, перед запуском дочернего процесса все дескрипторы файлов, кроме 0, 1 и 2, закрываются. По умолчанию используется значение False.
creation_flags	Определяет флаги создания процесса в Windows. В настоящее время доступен только один флаг – CREATE_NEW_CONSOLE. По умолчанию используется значение 0.
cwd	Каталог, в котором будет запущена команда. Перед запуском текущим рабочим каталогом дочернего процесса назначается cwd. По умолчанию используется значение None, которое соответствует использованию текущего рабочего каталога родительского процесса.
env	Словарь с переменными окружения для нового процесса. По умолчанию используется значение None, которое соответствует использованию окружения родительского процесса.
executable	Определяет имя выполняемой программы. Используется достаточно редко, так как имя программы уже включено в *args*. Если в этом параметре определить имя командной оболочки, команда будет запущена в этой командной оболочке. По умолчанию используется значение None.
preexec_fn	Определяет функцию в дочернем процессе, которая должна быть вызвана перед запуском команды. Функция не должна иметь входных аргументов.
shell	Если имеет значение True, команда выполняется в командной оболочке UNIX, с помощью функции os.system(). По умолчанию используется командная оболочка /bin/sh, но можно указать другую оболочку в параметре executable. По умолчанию используется значение None.
startupinfo	Определяет флаги запуска процесса в Windows. По умолчанию используется значение None. В число допустимых значений входят STARTF_USESHOWWINDOW и STARTF_USESTDHANDLERS.
stderr	Объект файла, который будет использоваться дочерним процессом как поток stderr. В этом параметре допускается передавать объект файла, созданный с помощью функции open(), целочисленный дескриптор файла или специальное значение PIPE, которое указывает на необходимость создания нового неименованного канала. По умолчанию используется значение None.
stdin	Объект файла, который будет использоваться дочерним процессом как поток stdin. В этом параметре допускается передавать те же значения, что и в параметре stderr. По умолчанию используется значение None.

Именованный аргумент	Описание
stdout	Объект файла, который будет использоваться дочерним процессом как поток stdout. В этом параметре допускается передавать те же значения, что и в параметре stderr. По умолчанию используется значение None.
universal_newlines	Если имеет значение True, файлы, представляющие потоки stdin, stdout и stderr, открываются в текстовом режиме с поддержкой универсального символа перевода строки. Подробнее об этом рассказывается в описании функции open().

`call(`*`args`*`, **`*`parms`*`)`

Эта функция выполняет те же действия, что и функция Popen(), за исключением того, что она просто выполняет команду и возвращает код завершения (то есть она не возвращает объект класса Popen). Эту функцию удобно использовать, когда требуется всего лишь выполнить команду и нет необходимости получать от нее вывод или управлять ею каким-либо способом. Аргументы имеют тот же смысл, что и в функции Popen().

`check_call(`*`args`*`, **`*`parms`*`)`

То же, что и call(), за исключением того, что в случае получения ненулевого кода завершения возбуждает исключение CalledProcessError. Код завершения сохраняется в атрибуте returncode исключения.

Объект *p* класса Popen, возвращаемый функцией Popen(), обладает различными методами и атрибутами, которые могут использоваться для взаимодействия с дочерним процессом.

p`.communicate([`*`input`*`])`

Передает данные *input* на стандартный ввод дочернего процесса. После отправки данных этот метод ожидает завершения дочернего процесса, продолжая принимать данные, которые выводятся дочерним процессом в его потоки стандартного вывода и стандартного вывода сообщений об ошибках. Возвращает кортеж (*stdout*, *stderr*), где поля *stdout* и *stderr* являются строками. Если не требуется передавать данные дочернему процессу, аргумент *input* можно установить в значение None (по умолчанию).

p`.kill()`

Принудительно завершает дочерний процесс, для чего в UNIX посылается сигнал SIGKILL, а в Windows вызывается метод *p*.terminate().

p`.poll()`

Проверяет, завершился ли дочерний процесс. Если процесс завершился, возвращает код завершения. В противном случае возвращает None.

p`.send_signal(`*`signal`*`)`

Посылает сигнал дочернему процессу. В аргументе *signal* передается номер сигнала, как определено в модуле signal. В Windows поддерживается единственный сигнал SIGTERM.

p.terminate()

Принудительно завершает дочерний процесс, посылая ему сигнал SIGTERM в UNIX или вызывая Win32 API функцию TerminateProcess в Windows.

p.wait()

Ожидает завершения дочернего процесса и возвращает код завершения.

p.pid

Целочисленный идентификатор дочернего процесса.

p.returncode

Код завершения дочернего процесса. Значение None свидетельствует о том, что дочерний процесс еще не завершился. Отрицательное значение указывает на то, что дочерний процесс завершился в результате получения сигнала (UNIX).

p.stdin, *p*.stdout, *p*.stderr

Эти три атрибута представляют открытые объекты файлов, соответствующие потокам ввода-вывода, открытым как неименованные каналы (например, установкой параметра stdout функции Popen() в значение PIPE). Эти объекты файлов обеспечивают возможность подключения к другим дочерним процессам. Эти атрибуты получают значение None, когда каналы не используются.

Примеры

```
# Выполнить простую системную команду с помощью os.system()
ret = subprocess.call("ls -l", shell=True)

# Выполнить простую команду, игнорируя все, что она выводит
ret = subprocess.call("rm -f *.java",shell=True,
                      stdout=open("/dev/null"))

# Выполнить системную команду, но сохранить ее вывод
p = subprocess.Popen("ls -l", shell=True, stdout=subprocess.PIPE)
out = p.stdout.read()

# Выполнить команду, передать ей входные данные и сохранить вывод
p = subprocess.Popen("wc", shell=True, stdin=subprocess.PIPE,
                     stdout=subprocess.PIPE, stderr=subprocess.PIPE)
out, err = p.communicate(s) # Передать строку s дочернему процессу

# Создать два дочерних процесса и связать их каналом
p1 = subprocess.Popen("ls -l", shell=True, stdout=subprocess.PIPE)
p2 = subprocess.Popen("wc",shell=True, stdin=p1.stdout,
                      stdout=subprocess.PIPE)
out = p2.stdout.read()
```

Примечания

Как правило, команды лучше передавать в виде списка строк, а не в виде единственной строки (например, ['wc','filename'] вместо 'wc filename').

Многие системы позволяют использовать символы пробела и другие неалфавитные символы в именах файлов (например, папка «Documents and Settings» в Windows). В подобных ситуациях, если команда определяется в виде списка строк, никаких проблем возникать не будет. Чтобы сформировать команду, где используются подобные имена файлов, вам потребуется выполнить дополнительные действия, чтобы экранировать специальные символы и пробелы.

В Windows каналы открываются в двоичном режиме. То есть текст, прочитанный из потока стандартного вывода дочернего процесса, будет содержать дополнительный символ возврата каретки на концах строк (`'\r\n'` вместо `'\n'`). Если это может вызывать проблемы, функции `Popen()` следует передавать значение `True` в параметре `universal_newlines`.

Модуль `subprocess` не может использоваться для управления дочерними процессами, которые предполагают возможность прямого доступа к терминалу или к устройству TTY. Наиболее типичными примерами таких программ являются любые команды, ожидающие ввода пароля пользователя (такие как `ssh`, `ftp`, `svn` и так далее). Для управления такими программами рекомендуется искать сторонние модули, основанные на популярной утилите «Expect», имеющейся в системе UNIX.

Модуль time

Модуль `time` предоставляет различные функции для работы со временем. В языке Python время измеряется количеством секунд, прошедших с начала эпохи. Начало *эпохи* соответствует началу исчисления времени (моменту, когда время было равно 0 секунд). В UNIX началу эпохи соответствует дата 1 января 1970 года, а в других системах может быть определено вызовом функции `time.gmtime(0)`.

Ниже перечислены переменные, объявленные в модуле:

`accept2dyear`

Логическое значение, указывающее – допускается ли использовать двузначное представление года. По умолчанию эта переменная имеет значение `True`, но она получит значение `False`, если в переменную окружения `$PYTHONY2K` записать непустую строку. Кроме того, значение переменной `accept2dyear` может изменяться вручную.

`altzone`

Часовой пояс, используемый после перехода на летнее время (DST), если применимо.

`daylight`

Имеет ненулевое значение, если определен часовой пояс, используемый после перехода на летнее время.

`timezone`

Локальный (не летний) часовой пояс.

`tzname`

Кортеж с названиями локального часового пояса и зимнего/летнего (если определен) часового пояса.

В модуле `time` также определены следующие функции:

`asctime([tuple])`

Преобразует кортеж, представляющий время, возвращаемое функцией `gmtime()` или `localtime()`, в строку вида `'Mon Jul 12 14:45:23 1999'`. При вызове без аргумента возвращает представление текущего времени.

`clock()`

Возвращает текущее процессорное время в секундах в виде числа с плавающей точкой.

`ctime([secs])`

Преобразует время *secs*, выраженное в секундах от начала эпохи, в строку, представляющую локальное время. Вызов `ctime(secs)` **эквивалентен вызову** `asctime(localtime(secs))`. При вызове без аргумента или со значением `None` в аргументе возвращает представление текущего времени.

`gmtime([secs])`

Принимает время *secs*, выраженное в секундах от начала эпохи по Гринвичскому времени (Coordinated Universal Time, UTC), и возвращает объект типа `struct_time`, обладающий следующими атрибутами:

Атрибут	Значение
`tm_year`	Четырехзначный год, например `1998`
`tm_mon`	`1-12`
`tm_mday`	`1-31`
`tm_hour`	`0-23`
`tm_min`	`0-59`
`tm_sec`	`0-61`
`tm_wday`	`0-6` (`0` = понедельник)
`tm_yday`	`1-366`
`tm_isdst`	`-1`, `0`, `1`

Когда действует летнее время (DST), атрибут `tm_isdst` принимает значение `1`, `0` – если действует зимнее время, и `-1` – если эта информация недоступна. При вызове без аргумента или когда в аргументе *secs* передается значение `None`, используется текущее время. Для обратной совместимости возвращаемый объект `struct_time` может вести себя как кортеж с девятью значениями, описанными выше, следующими в том же порядке.

`localtime([secs])`

Возвращает объект типа `struct_time`, как и функция `gmtime()`, информация в котором соответствует локальному часовому поясу. При вызове без аргу-

мента или когда в аргументе *secs* передается значение None, используется текущее время.

`mktime(`*tuple*`)`

Принимает объект типа struct_time или кортеж, представляющий время для локального часового пояса (в том же формате, в каком оно возвращается функцией localtime()), и возвращает число с плавающей точкой, представляющее количество секунд, прошедших от начала эпохи. Если в аргументе *tuple* передается недопустимое значение, возбуждает исключение OverflowError.

`sleep(`*secs*`)`

Приостанавливает работу текущего процесса на *secs* секунд. В аргументе *secs* передается число с плавающей точкой.

`strftime(`*format* `[,` *tm*`])`

Преобразует объект *tm* типа struct_time, представляющий время в том же формате, в каком оно возвращается функцией localtime() или gmtime(), в строку (для обратной совместимости функция strftime() может принимать в аргументе *tm* кортеж, представляющий время). В аргументе *format* передается строка формата, в которой могут использоваться следующие спецификаторы формата:

Атрибут	Значение
%a	Сокращенное название дня недели
%A	Полное название дня недели
%b	Сокращенное название месяца
%B	Полное название месяца
%c	Представление даты и времени в соответствии с региональными настройками
%d	Десятичное представление числа месяца [01-31]
%H	Десятичное представление часа (в 24-часовой шкале) [00-23]
%I	Десятичное представление часа (в 12-часовой шкале) [01-12]
%j	Десятичное представление дня года [001-366]
%m	Десятичное представление месяца [01-12]
%M	Десятичное представление минут [00-59]
%p	Национальный эквивалент обозначения AM (до полудня) и PM (после полудня)
%S	Десятичное представление секунд [00-61]
%U	Номер недели года [00-53] (считается, что неделя начинается в воскресенье, все дни в году до первого воскресенья относятся к неделе с номером 00)
%w	Десятичное представление дня недели (0 = воскресенье)

(продолжение)

Атрибут	Значение
%W	Номер недели года [00-53] (считается, что неделя начинается в понедельник, все дни в году до первого воскресенья относятся к неделе с номером 00)
%x	Представление даты в соответствии с региональными настройками
%X	Представление времени в соответствии с региональными настройками
%y	Десятичное представление года без указания века [00-99]
%Y	Полное десятичное представление года
%Z	Название часового пояса (или пустая строка, если часовой пояс не определен)
%%	Символ %

Спецификаторы формата могут включать значения ширины поля вывода и точности представления, как это делается при использовании оператора % форматирования строк. При выходе какого-либо значения в кортеже за допустимый диапазон возбуждается исключение ValueError. При вызове без аргумента используется текущее время.

`strptime(`*`string`* `[,` *`format`*`])`

Анализирует строку *string* с представлением времени и возвращает объект типа struct_time в том же формате, в каком его возвращают функции localtime() и gmtime(). В строке формата *format* допускается использовать те же спецификаторы, которые используются в функции strftime(). По умолчанию аргумент *format* принимает значение '%a %b %d %H:%M:%S %Y'. Этот же формат используется функцией ctime(). В случае обнаружения ошибок в строке *string* возбуждает исключение ValueError.

`time()`

Возвращает текущее время по Гринвичу (Coordinated Universal Time, UTC) в виде количества секунд, прошедших от начала эпохи.

`tzset()`

Устанавливает часовой пояс, опираясь на значение переменной окружения TZ в UNIX. Например:

```
os.environ['TZ'] = 'US/Mountain'
time.tzset()

os.environ['TZ'] = "CST+06CDT,M4.1.0,M10.5.0"
time.tzset()
```

Примечания

Если допускается указывать год в виде двухзначного числа, это число будет преобразовано в четырехзначное, в соответствии с требованиями стандар-

та POSIX X/Open, согласно которым значения 69-99 преобразуются в 1969-1999, а значения 0-68 — в 2000-2068.

Точность представления времени в функциях может оказаться хуже, чем можно было бы предположить, исходя из единиц представления, с которыми они работают. Например, операционная система может обновлять внутренние часы всего 50-100 раз в секунду.

См. также

Описание модуля datetime (стр. 421).

Модуль winreg

Модуль winreg (в Python 2 он называется _winreg) предоставляет низкоуровневый интерфейс к реестру Windows. Реестр — это крупное хранилище данных с древовидной структурой, где каждый узел называется *ключом*. Вложенные записи, которые называют *подключами*, могут содержать значения или другие подключи. Например, значение переменной sys.path обычно хранится в реестре в следующем виде:

```
\HKEY_LOCAL_MACHINE\Software\Python\PythonCore\2.6\PythonPath
```

В этом примере Software является подключом ключа HKEY_LOCAL_MACHINE, Python — подключом ключа Software и так далее. Ключ PythonPath хранит фактическое значение переменной sys.path.

Доступ к ключам осуществляется с помощью операций открытия и закрытия. Открытые ключи представлены специальными дескрипторами (которые представляют собой обертки вокруг целочисленных значений дескрипторов, обычно используемых в Windows).

`CloseKey(`*`key`*`)`

Закрывает ранее открытый ключ реестра с дескриптором *key*.

`ConnectRegistry(`*`computer_name`*`, `*`key`*`)`

Возвращает дескриптор предопределенного ключа реестра на удаленном компьютере. В аргументе *computer_name* передается строка с именем удаленного компьютера, имеющая вид \\computername. Если в аргументе computer_name передать значение None, будет использоваться локальный реестр. В аргументе *key* передается дескриптор предопределенного ключа, такого как HKEY_CURRENT_USER или HKEY_ USERS. В случае ошибки возбуждает исключение EnvironmentError. Ниже перечислены все значения HKEY_*, объявленные в модуле _winreg:

- HKEY_CLASSES_ROOT
- HKEY_CURRENT_CONFIG
- HKEY_CURRENT_USER
- HKEY_DYN_DATA
- HKEY_LOCAL_MACHINE

- HKEY_PERFORMANCE_DATA
- HKEY_USERS

`CreateKey(key, sub_key)`

Создает или открывает ключ и возвращает его дескриптор. В аргументе *key* передается ранее открытый ключ или один из предопределенных ключей HKEY_*. В аргументе *sub_key* передается имя ключа, который требуется открыть или создать. Если аргумент *key* представляет предопределенный ключ, в аргументе *sub_key* допускается передавать значение None; в этом случае будет возвращен дескриптор ключа *key*.

`DeleteKey(key, sub_key)`

Удаляет подключ *sub_key*. В аргументе *key* передается ранее открытый ключ или один из предопределенных ключей HKEY_*. В аргументе *sub_key* передается строка, идентифицирующая удаляемый ключ. Подключ *sub_key* не должен содержать вложенные ключи; в противном случае будет возбуждено исключение EnvironmentError.

`DeleteValue(key, value)`

Удаляет из ключа реестра значение с именем *value*. В аргументе *key* передается ранее открытый ключ или один из предопределенных ключей HKEY_*. В аргументе *value* передается строка с именем удаляемого значения.

`EnumKey(key, index)`

Возвращает имя подключа по индексу *index*. В аргументе *key* передается ранее открытый ключ или один из предопределенных ключей HKEY_*. В аргументе *index* передается целое число, определяющее требуемый ключ. Если значение *index* выходит за пределы допустимого диапазона, возбуждает исключение EnvironmentError.

`EnumValue(key, index)`

Возвращает значение открытого ключа. В аргументе *key* передается ранее открытый ключ или один из предопределенных ключей HKEY_*. В аргументе *index* передается целое число, определяющее требуемое значение. Функция возвращает кортеж (*name, data, type*), где поле *name* содержит имя значения, *data* – объект, представляющий значение, и *type* – целое число, определяющее тип значения. Ниже приводится перечень возможных значений поля *type*, определенных в настоящее время:

Код	Значение
REG_BINARY	Двоичные данные
REG_DWORD	32-битное число
REG_DWORD_LITTLE_ENDIAN	32-битное число с обратным порядком следования байтов
REG_DWORD_BIG_ENDIAN	32-битное число с прямым порядком следования байтов

Код	Значение
REG_EXPAND_SZ	Строка, оканчивающаяся символом '\0' и содержащая ссылки на имена переменных окружения
REG_LINK	Символическая ссылка с символами Юникода
REG_MULTI_SZ	Последовательность строк, оканчивающихся символом '\0'.
REG_NONE	Значение неопределенного типа
REG_RESOURCE_LIST	Список ресурсов драйвера устройства
REG_SZ	Строка, оканчивающаяся символом '\0'.

`ExpandEnvironmentStrings(`*s*`)`

Выполняет замену имен переменных окружения в строке Юникода *s*, указанных в виде `%name%`, их значениями.

`FlushKey(`*key*`)`

Записывает в реестр атрибуты ключа *key*, принудительно сохраняет изменения на диске. Эта функция должна вызываться, только если имеется полная уверенность, что реестр хранится на диске. Возвращает управление, только когда данные будут записаны. В обычных обстоятельствах эту функцию желательно не использовать.

`RegLoadKey(`*key*, *sub_key*, *filename*`)`

Создает подключ и сохраняет в нем информацию из файла. В аргументе *key* передается ранее открытый ключ или один из предопределенных ключей `HKEY_*`. В аргументе *sub_key* передается строка, идентифицирующая подключ, куда будут загружены данные. В аргументе *filename* передается имя файла, откуда будут загружаться данные. Содержимое этого файла должно быть создано функцией `SaveKey()`, а вызывающий процесс должен обладать привилегией `SE_RESTORE_PRIVILEGE`. Если ключ был получен вызовом функции `ConnectRegistry()`, строка *filename* должна представлять путь файлу относительно удаленного компьютера.

`OpenKey(`*key*, *sub_key*`[, `*res*` [, `*sam*`]])`

Открывает ключ. В аргументе *key* передается ранее открытый ключ или один из предопределенных ключей `HKEY_*`. В аргументе *sub_key* передается строка, идентифицирующая открываемый подключ. Аргумент *res* – зарезервированное целое число, которое должно быть нулем (по умолчанию). В аргументе *sam* передается целое число, определяющее маску прав доступа к ключу. По умолчанию принимает значение `KEY_READ`. Ниже перечислены возможные значения аргумента *sam*:

- `KEY_ALL_ACCESS`
- `KEY_CREATE_LINK`
- `KEY_CREATE_SUB_KEY`
- `KEY_ENUMERATE_SUB_KEYS`

- KEY_EXECUTE
- KEY_NOTIFY
- KEY_QUERY_VALUE
- KEY_READ
- KEY_SET_VALUE
- KEY_WRITE

`OpenKeyEx()`

То же, что и `OpenKey()`.

`QueryInfoKey(`*`key`*`)`

Возвращает информацию о ключе в виде кортежа (*num_subkeys*, *num_values*, *last_modified*), где в поле *num_subkeys* возвращается количество подключей, в поле *num_values* – количество значений и в поле *last_modified* – длинное целое число, содержащее время последнего изменения. Время измеряется в сотнях наносекунд, прошедших с 1 января 1601 года.

`QueryValue(`*`key`*`,`*`sub_key`*`)`

Возвращает неименованное значение ключа в виде строки. В аргументе *key* передается ранее открытый ключ или один из предопределенных ключей `HKEY_*`. В аргументе *sub_key* передается имя подключа, если необходимо. При вызове без аргумента *sub_key* функция возвращает значение, ассоциированное с самим ключом *key*. Возвращается первое значение с пустым именем. Однако тип значения не возвращается (чтобы получить информацию о типе значения, следует использовать функцию `QueryValueEx`).

`QueryValueEx(`*`key`*`, `*`value_name`*`)`

Возвращает кортеж (*value*, *type*) со значением ключа и информацией о типе этого значения. В аргументе *key* передается ранее открытый ключ или один из предопределенных ключей `HKEY_*`. В аргументе *value_name* передается имя требуемого значения. В качестве типа возвращается одно из целочисленных значений, которые были перечислены в описании функции `EnumValue()`.

`SaveKey(`*`key`*`, `*`filename`*`)`

Сохраняет ключ *key* и все вложенные подключи в файле. В аргументе *key* передается ранее открытый ключ или один из предопределенных ключей `HKEY_*`. В аргументе *filename* должно передаваться имя несуществующего файла без расширения. Кроме того, вызывающий процесс должен обладать правом на резервное копирование.

`SetValue(`*`key`*`, `*`sub_key`*`, `*`type`*`, `*`value`*`)`

Устанавливает значение ключа. В аргументе *key* передается ранее открытый ключ или один из предопределенных ключей `HKEY_*`. В аргументе *sub_key* передается имя подключа, в который будет записано значение. В аргументе *type* передается целочисленный код типа значения, причем в настоящее время допускается использовать только тип `REG_SZ`. В аргументе *value* передается строка, содержащая значение. Если подключ *sub_key* не суще-

ствует, он будет создан. Для выполнения этой операции ключ *key* должен быть открыт с правами доступа KEY_SET_VALUE.

SetValueEx(*key*, *value_name*, *reserved*, *type*, *value*)

Устанавливает значение ключа. В аргументе *key* передается ранее открытый ключ или один из предопределенных ключей HKEY_*. В аргументе *value_name* передается имя значения. В аргументе *type* передается целочисленный код типа значения, из числа тех, что перечислены в описании функции EnumValue(). В аргументе *value* передается строка, содержащая новое значение. Даже при передаче числовых значений (например, REG_DWORD) аргумент *value* все равно должен быть строкой, содержащей двоичные данные. Такую строку можно создать с помощью модуля struct. Аргумент *reserved* в настоящее время игнорируется и может принимать любые значения (значение этого аргумента никак не используется).

Примечания

- Функции, возвращающие объект **HKEY ключа реестра Windows, возвращают** специальный объект дескриптора класса PyHKEY. Этот объект может быть преобразован в дескриптор, используемый функциями Win32 API, с помощью функции int(). Кроме того, объекты этого класса поддерживают протокол менеджеров контекста, что, например, позволяет автоматически закрывать дескриптор:

```
with winreg.OpenKey(winreg.HKEY_LOCAL_MACHINE, "spam") as key:
    инструкции
```

20

Потоки и многозадачность

В этой главе описываются модули и приемы программирования, используемые при разработке многопоточных приложений на языке Python. В число рассматриваемых тем входят: потоки управления, обмен сообщениями, многопоточная обработка данных и сопрограммы. Прежде чем перейти к рассмотрению библиотечных модулей, познакомимся с некоторыми основными понятиями.

Основные понятия

Выполняющаяся программа называется *процессом*. Каждый процесс обладает некоторыми параметрами, характеризующими его состояние, включая объем занимаемой памяти, список открытых файлов, программный счетчик, который ссылается на очередную выполняемую инструкцию, и стек вызовов, используемый для хранения локальных переменных функций. Обычно процесс выполняет инструкции одну за другой, в единственном потоке управления, который иногда называют *главным потоком* процесса. В каждый конкретный момент программа делает что-то одно.

Программа может создавать новые процессы с помощью библиотечных функций, таких как представленные в модуле `os` или `subprocess` (например, `os.fork()`, `subprocess.Popen()` и другие). Такие процессы называют *дочерними процессами*. Они выполняются совершенно независимо, и каждый из них имеет свои собственные характеристики и главный поток управления. Так как дочерние процессы являются независимыми, они выполняются параллельно с родительским процессом. То есть родительский процесс, создавший дочерние процессы, может выполнять свои операции, тогда как дочерние процессы будет за кулисами выполнять свою работу.

Несмотря на то что процессы изолированы друг от друга, тем не менее они могут обмениваться информацией, используя механизмы *взаимодействия процессов* (Interprocess Communication, IPC). Одной из распространенных форм взаимодействия процессов является *обмен сообщениями*. Под сообщением в данном случае понимается простой буфер двоичных

байтов. Для приема и передачи сообщений через каналы ввода-вывода, например неименованные каналы или сетевые соединения, используются простейшие операции, такие как `send()` и `recv()`. Другой, реже используемый механизм взаимодействий, основан на использовании отображаемых областей памяти (смотрите описание модуля `mmap`). Благодаря возможности отображения в память процессы могут создавать разделяемые области памяти. Изменения в этих областях будут доступны всем процессам, имеющим к ним доступ.

Возможность создания дочерних процессов может использоваться приложениями для выполнения сразу нескольких задач, когда каждый процесс отвечает за свою часть работы. Однако существует и другой подход к разделению работы на задачи, основанный на использовании нескольких потоков управления. *Поток управления* напоминает процесс тем, что он выполняет собственную последовательность инструкций и имеет свой стек вызовов. Разница состоит в том, что потоки выполняются в пределах процесса, создавшего их, и они совместно используют данные и системные ресурсы, выделенные процессу. Потоки удобно использовать, когда в приложении необходимо реализовать одновременное выполнение нескольких задач, но при этом имеется значительный объем информации, которая должна быть доступна всем потокам.

Когда одновременно выполняется несколько процессов или потоков управления, за распределение процессорного времени между ними отвечает операционная система. Она выделяет каждому процессу (или потоку) небольшой квант времени и быстро переключается между активными задачами, отдавая каждой из них определенное количество тактов процессора. Например, если в системе имеется 10 активных процессов, операционная система будет выделять каждому из них примерно 1/10 процессорного времени и будет быстро переключаться между ними. В системах, где имеется более одного ядра процессора, операционная система сможет планировать выполнение процессов так, чтобы все ядра процессоров оказались заняты примерно поровну параллельным выполнением процессов.

Разработка программ, использующих преимущества параллельного выполнения задач, является несомненно сложным делом. Основная сложность заключается в синхронизации и обеспечении доступа к совместно используемым данным. В частности, попытка изменить некоторые данные одновременно из нескольких потоков может привести к их повреждению и к нарушению целостности состояния программы (формально эта проблема известна как *гонка за ресурсами*). Чтобы избавиться от этой проблемы, в многопоточных программах необходимо выделить критические участки программного кода и обеспечить их выполнение под защитой взаимоисключающих блокировок или других, похожих механизмов синхронизации. Например, если в разных потоках одновременно появится необходимость выполнить запись в один и тот же файл, для синхронизации их действий можно воспользоваться взаимоисключающей блокировкой, чтобы, пока один из потоков выполняет запись, остальные ожидали бы завершения операции и только после этого получали доступ к файлу. Реализация синхронизации подобного рода обычно выглядит, как показано ниже:

```
write_lock = Lock()
...
# Критический участок, где выполняется запись
write_lock.acquire()
f.write("Here's some data.\n")
f.write("Here's more data.\n")
...
write_lock.release()
```

Существует шутка, которая приписывается Джейсону Уиттингтону (Jason Whittington): «Зачем многопоточный цыпленок пересекает дорогу? на Чтобы другую сторону. перейти». Эта шутка символизирует типичные проблемы, связанные с синхронизацией и многопоточным программированием. Если вы в задумчивости почесываете затылок и говорите себе: «Я ничего не понял», – то вам стоит почитать дополнительную литературу, прежде чем продолжать чтение этой главы.

Параллельное программирование и Python

Python **обеспечивает поддержку механизма обмена сообщениями и позволяет** создавать многопоточные программы в большинстве систем. Большинство программистов знакомы с интерфейсом потоков, но не многие знают, что потоки управления в языке **Python имеют существенные ограничения.** Несмотря на наличие минимальной поддержки многопоточных приложений, интерпретатор Python использует внутреннюю глобальную блокировку интерпретатора (Global Interpreter Lock, GIL), из-за чего в каждый конкретный момент времени может выполняться только один поток. Вследствие этого программы на языке Python могут выполняться только на одном процессоре, независимо от их количества в системе. Наличие глобальной блокировки **GIL часто становится источником жарких дебатов в сообществе Python, тем не менее весьма маловероятно, что она будет устранена** в обозримом будущем.

Присутствие блокировки **GIL напрямую влияет на количество программистов,** занимающихся проблемами разработки многопоточных приложений. Вообще говоря, если приложение в основном занимается операциями ввода-вывода, оно, как правило, является прекрасным кандидатом на реализацию в виде многопоточного приложения, потому что наличие дополнительных процессоров практически не дает преимуществ программе, которая большую часть времени проводит в ожидании событий. С другой стороны, разделение приложения, выполняющего объемные вычисления, на несколько потоков не только не дает никаких преимуществ, но и снизит производительность программы (причем снижение производительности окажется *намного существеннее*, чем вы могли бы предположить). Подобные программы лучше реализовать в виде нескольких процессов и обеспечить взаимодействие между ними с привлечением механизма обмена сообщениями.

Даже те программисты, которые используют механизм потоков, зачастую находят весьма странными их поведение при масштабировании. Например, многопоточный сетевой сервер может иметь прекрасную производи-

тельность при одновременной работе 100 потоков и просто ужасную, когда их число увеличивается до 10 000. Вообще говоря, не стоит писать программы, запускающие 10 000 потоков, потому что каждый поток потребляет ресурсы системы и увеличивает нагрузку, связанную с необходимостью переключения контекста потоков, установки блокировок и использования других механизмов, интенсивное использование которых приводит к увеличению нагрузки (не говоря уже о том, что все потоки вынуждены выполняться на единственном процессоре). Подобные проблемы часто решаются за счет реструктуризации приложения и использования систем асинхронной обработки событий. Например, главный цикл событий может просматривать все каналы ввода-вывода, используя модуль `select`, и передавать асинхронные события большой коллекции обработчиков ввода-вывода. Подобный подход был положен в основу некоторых библиотечных модулей, таких как `asyncore`, а также популярных сторонних модулей, таких как `Twisted` (*http://twistedmatrix/com*).

Забегая вперед, скажу, что, приступая к разработке многопоточных приложений на языке Python, в первую очередь необходимо выбрать механизм обмена сообщениями. При работе с потоками часто рекомендуется организовать приложение как коллекцию независимых потоков, обменивающихся данными с помощью очередей сообщений. Такой подход меньше подвержен ошибкам, потому что он снижает необходимость использования блокировок и других механизмов синхронизации. Кроме того, обмен сообщениями легко и естественно распространяется на сетевые взаимодействия и распределенные системы. Например, если имеется часть программы, которая выполняется в виде отдельного потока и принимает сообщения, эту часть затем можно оформить в виде отдельного процесса или перенести на другой компьютер и посылать ей сообщения через сетевое соединение. Кроме того, абстракция механизма обмена сообщениями легко совмещается с такими понятиями языка Python, как сопрограммы. Например, сопрограмма – это функция, которая принимает и обрабатывает сообщения, передаваемые ей. Поэтому, приняв за основу механизм обмена сообщениями, вы обнаружите, что в состоянии писать программы, обладающие существенной гибкостью.

В оставшейся части главы будут рассматриваться различные библиотечные модули, обеспечивающие поддержку многопоточного программирования. В конце главы приводится более подробная информация о наиболее типичных идиомах программирования.

Модуль multiprocessing

Модуль `multiprocessing` предоставляет поддержку запуска задач в виде дочерних процессов, взаимодействий между ними и совместного использования данных, а также обеспечивает различные способы синхронизации. Программный интерфейс модуля имитирует программный интерфейс потоков, реализованный в модуле `threading`. Однако важное отличие от потоков состоит в том, что процессы не имеют совместно используемых данных. То есть, если процесс изменит данные, эти изменения будут носить локальный характер для данного процесса.

Модуль multiprocessing обладает весьма широкими возможностями, что делает его одной из самых крупных и самых сложных встроенных библиотек. Здесь невозможно во всех подробностях описать каждую особенность модуля, тем не менее мы рассмотрим самые основные его части, наряду с примерами использования. Опытные программисты смогут взять за основу эти примеры и распространить используемые в них приемы на более сложные задачи.

Процессы

Все функциональные возможности, имеющиеся в модуле multiprocessing, направлены на работу с процессами, которые описываются следующим классом.

Process ([*group* [, *target* [, *name* [, *args* [, *kwargs*]]]]])

Класс, представляющий задачу, запущенную в дочернем процессе. Параметры всегда должны передаваться конструктору в виде именованных аргументов. В аргументе *target* передается объект, поддерживающий возможность вызова, который будет выполнен при запуске процесса, в аргументе *args* передается кортеж позиционных аргументов для функции *target*, а в аргументе *kwargs* – словарь именованных аргументов для объекта *target*. Если опустить аргументы *args* и *kwargs*, объект *target* будет вызван без аргументов. В аргументе *name* передается строка с описательным именем процесса. Аргумент *group* не используется и всегда принимает значение None. Он присутствует лишь для полноты имитации создания потоков с помощью модуля threading.

Экземпляр *p* класса Process обладает следующими методами:

p.is_alive()

Возвращает True, если процесс *p* продолжает работу.

p.join([*timeout*])

Ожидает завершения процесса *p*. Аргумент *timeout* определяет максимальный период ожидания. Присоединяться к процессу, в ожидании его завершения, можно неограниченное число раз, но будет ошибкой, если процесс попытается присоединиться к себе самому.

p.run()

Метод, который вызывается в дочернем процессе при его запуске. По умолчанию вызывает объект *target*, который был передан конструктору класса Process. При желании можно создать свой класс, производный от класса Process, и определить в нем собственную реализацию метода run().

p.start()

Запускает процесс. Запускает дочерний процесс и вызывает в нем метод *p*.run().

p.terminate()

Принудительно завершает процесс. При вызове этого метода дочерний процесс завершается немедленно, без выполнения каких-либо заключительных процедур. Если процесс *p* создал свои дочерние процессы, они превра-

тятся в «зомби». Этот метод требует осторожного обращения. Если процесс *p* установил блокировку или вовлечен во взаимодействия с другими процессами, его принудительное завершение может вызвать взаимоблокировку процессов или повреждение данных.

Экземпляр *p* класса `Process` обладает также следующими атрибутами:

p.`authkey`

Ключ аутентификации процесса. Если значение не было определено явно, в этот атрибут записывается 32-символьная строка, сгенерированная функцией `os.urandom()`. Назначение этого ключа состоит в том, чтобы обеспечить безопасность низкоуровневых операций взаимодействия между процессами, которые выполняются через сетевые соединения. Взаимодействия через такие соединения будут возможны, только если с обоих концов используется один и тот же ключ аутентификации.

p.`daemon`

Логический флаг, указывающий – будет ли дочерний процесс демоническим. *Демонический*[1] процесс завершается автоматически, вместе с процессом Python, создавшим его. Кроме того, демонический процесс лишен возможности создавать дочерние процессы. Значение атрибута *p*.`daemon` должно устанавливаться до вызова метода *p*.`start()`.

p.`exitcode`

Целочисленный код завершения процесса. Если процесс продолжает выполняться, этот атрибут будет содержать значение `None`. Отрицательное значение *-N* означает, что процесс был завершен сигналом *N*.

p.`name`

Имя процесса.

p.`pid`

Целочисленный идентификатор процесса.

Следующий пример демонстрирует, как создавать и запускать функции (или другие вызываемые объекты) в отдельном процессе:

```
import multiprocessing
import time

def clock(interval):
    while True:
        print("The time is %s" % time.ctime())
        time.sleep(interval)

if __name__ == '__main__':
    p = multiprocessing.Process(target=clock, args=(15,))
    p.start()
```

[1] Здесь понятие *демонический* не имеет никакого отношения к понятию *демона* (фонового процесса) в UNIX. Под демоническим процессом понимается процесс, который просто автоматически завершается вместе с родительским процессом. – *Прим. перев.*

Следующий пример демонстрирует, как определить свой класс, производный от класса Process:

```
import multiprocessing
import time

class ClockProcess(multiprocessing.Process):
    def __init__(self,interval):
        multiprocessing.Process.__init__(self)
        self.interval = interval
    def run(self):
        while True:
            print("The time is %s" % time.ctime())
            time.sleep(self.interval)

if __name__ == '__main__':
    p = ClockProcess(15)
    p.start()
```

В обоих примерах дочерние процессы должны выводить текущее время каждые 15 секунд. Важно заметить, что для обеспечения межплатформенной совместимости новые процессы должны создаваться только основной программой, как показано в примерах. В UNIX это условие можно не соблюдать, но в Windows оно является обязательным. Кроме того, нужно отметить, что в Windows предыдущие примеры необходимо запускать в командной оболочке (command.exe), а не в интегрированной среде разработки на языке Python, такой как IDLE.

Взаимодействия между процессами

Модуль multiprocessing поддерживает два основных способа взаимодействия процессов: каналы и очереди. Оба способа реализованы на основе механизма передачи сообщений. Причем интерфейс очередей очень близко имитирует интерфейс очередей, которые обычно используются в многопоточных программах.

`Queue([maxsize])`

Создает очередь для организации обмена сообщениями между процессами. Аргумент *maxsize* определяет максимальное количество элементов, которые можно поместить в очередь. При вызове без аргумента размер очереди не ограничивается. Внутренняя реализация очередей основана на использовании каналов и блокировок. Кроме того, для передачи данных из очереди в канал запускается вспомогательный поток управления.

Экземпляр *q* класса Queue обладает следующими методами:

`q.cancel_join_thread()`

Предотвращает автоматическое присоединение к фоновому потоку при завершении процесса. Благодаря этому исключается возможность блокирования процесса в вызове метода join_thread().

`q.close()`

Закрывает очередь, запрещая добавление новых элементов. После вызова этого метода фоновый вспомогательный поток продолжит запись данных

из очереди в канал и завершится, как только очередь будет исчерпана. Этот метод вызывается автоматически, когда экземпляр *q* утилизируется сборщиком мусора. Операция закрытия очереди не генерирует какой-либо признак окончания передачи данных и не возбуждает исключение на принимающей стороне. Например, если принимающий процесс (потребитель) находится в ожидании в методе `get()`, закрытие очереди на стороне передающего процесса (поставщика) не повлечет выход из метода `get()`с признаком ошибки на стороне потребителя.

q.`empty()`

Возвращает `True`, если в момент вызова очередь *q* была пустой. Имейте в виду, что если имеются другие процессы и потоки, добавляющие новые элементы в очередь, результат вызова этой функции не может считаться надежным (между моментом получения результата и моментом его проверки в очередь могут быть добавлены новые элементы).

q.`full()`

Возвращает `True`, если в момент вызова очередь *q* была полной. Результат этой функции также нельзя считать надежным в многопоточных приложениях (смотрите описание *q*.`empty()`).

q.`get([`*block* `[, ` *timeout*`]])`

Возвращает элемент из очереди *q*. Если очередь *q* пуста, процесс приостанавливается до появления элемента в очереди. Аргумент *block* управляет режимом блокировки и по умолчанию имеет значение `True`. Если в этом аргументе передать `False`, при попытке получить элемент из пустой очереди метод будет возбуждать исключение `Queue.Empty` (объявляется в библиотечном модуле `Queue`). Аргумент *timeout* является необязательным и используется, когда блокировка разрешена. Если в течение указанного интервала времени в очереди не появится сообщений, будет возбуждено исключение `Queue.Empty`.

q.`get_nowait()`

То же, что и *q*.`get(False)`.

`q.join_thread()`

Выполняет присоединение к фоновому потоку очереди. Может использоваться, чтобы дождаться момента, когда очередь будет исчерпана после вызова метода *q*.`close()`. По умолчанию этот метод вызывается всеми процессами, которые не являются создателями очереди *q*. Такое поведение можно изменить, вызвав метод *q*.`cancel_join_thread()`.

q.`put(item [, ` *block* `[, ` *timeout*`]])`

Добавляет элемент *item* в очередь. Если очередь заполнена до предела, процесс приостанавливается до появления места в очереди. Аргумент *block* управляет режимом блокировки и по умолчанию имеет значение `True`. Если в этом аргументе передать `False`, при попытке добавить элемент в заполненную очередь метод будет возбуждать исключение `Queue.Full` (объявляется в библиотечном модуле `Queue`). Аргумент *timeout* определяет предельное время ожидания появления свободного места в очереди, когда блокировка

разрешена. По истечении времени ожидания будет возбуждено исключение `Queue.Full`.

`q.put_nowait(item)`

То же, что и `q.put(item, False)`.

`q.qsize()`

Возвращает примерное количество элементов, находящихся в очереди. Результат этой функции также нельзя считать надежным, потому что между моментом получения результата и моментом его проверки могут быть добавлены новые элементы или извлечены существующие. В некоторых системах этот метод может возбуждать исключение `NotImplementedError`.

`JoinableQueue([maxsize])`

Создает необособленный процесс очереди, доступной для совместного использования. Очереди этого типа очень похожи на очереди типа `Queue`, за исключением того, что очередь позволяет потребителю известить поставщика, что элементы были благополучно обработаны. Процедура передачи извещений реализована на основе разделяемых семафоров и переменных состояния.

Экземпляр `q` класса `JoinableQueue` обладает теми же методами, что и экземпляр класса `Queue`, и сверх того имеет следующие дополнительные методы:

`q.task_done()`

Используется потребителем, чтобы сообщить, что элемент очереди, полученный методом `q.get()`, был обработан. Возбуждает исключение `ValueError`, если количество вызовов этого метода превышает количество элементов, извлеченных из очереди.

`q.join()`

Используется поставщиком, чтобы дождаться момента, когда будут обработаны все элементы очереди. Этот метод приостанавливает процесс, пока для каждого элемента в очереди не будет вызван метод `q.task_done()`.

Следующий пример демонстрирует, как реализовать процесс, который в бесконечном цикле получает элементы из очереди и обрабатывает их. Поставщик добавляет элементы в очередь и ожидает, пока они не будут обработаны.

```
import multiprocessing

def consumer(input_q):
    while True:
        item = input_q.get()
        # Обработать элемент
        print(item)          # Заместите эту инструкцию фактической обработкой
        # Сообщить о завершении обработки
        input_q.task_done()

def producer(sequence, output_q):
    for item in sequence:
        # Добавить элемент в очередь
        output_q.put(item)
```

```
# Настройка
if __name__ == '__main__':
    q = multiprocessing.JoinableQueue()
    # Запустить процесс-потребитель
    cons_p = multiprocessing.Process(target=consumer,args=(q,))
    cons_p.daemon=True
    cons_p.start()

    # Воспроизвести элементы.
    # Переменная sequence представляет последовательность элементов, которые
    # будут передаваться потребителю. На практике вместо переменной можно
    # использовать генератор или воспроизводить элементы каким-либо другим
    # способом.
    sequence = [1,2,3,4]
    producer(sequence, q)

    # Дождаться, пока все элементы не будут обработаны
    q.join()
```

В этом примере процесс-потребитель запускается как демонический процесс, потому что он выполняется в бесконечном цикле, а нам необходимо, чтобы он завершался вместе с главной программой (если этого не сделать, программа зависнет). Чтобы иметь возможность в процессе-поставщике определить момент, когда все элементы будут успешно обработаны, используется очередь типа `JoinableQueue`**. Это гарантирует метод** `join()`**; если забыть вызвать этот метод, процесс-потребитель будет завершен еще до того, как успеет обработать все элементы в очереди.**

Добавлять и извлекать элементы очереди могут сразу несколько процессов. Например, если данные должны получать сразу несколько процессов-потребителей, это можно было бы реализовать, как показано ниже:

```
if __name__ == '__main__':
    q = multiprocessing.JoinableQueue()
    # Запустить несколько процессов-потребителей
    cons_p1 = multiprocessing.Process(target=consumer,args=(q,))
    cons_p1.daemon=True
    cons_p1.start()

    cons_p2 = multiprocessing.Process(target=consumer,args=(q,))
    cons_p2.daemon=True
    cons_p2.start()

    # Воспроизвести элементы.
    # Переменная sequence представляет последовательность элементов, которые
    # будут передаваться потребителю. На практике вместо переменной можно
    # использовать генератор или воспроизводить элементы каким-либо другим
    # способом.
    sequence = [1,2,3,4]
    producer(sequence, q)

    # Дождаться, пока все элементы не будут обработаны
    q.join()
```

При разработке подобного программного кода не забывайте, что каждый элемент, помещаемый в очередь, преобразуется в последовательность байтов и отправляется процессу через канал или сетевое соединение. Как правило, с точки зрения производительности лучше послать небольшое количество крупных объектов, чем много маленьких.

В некоторых ситуациях бывает желательно, чтобы поставщик извещал потребителей, что элементов больше не будет и потребители должны завершить работу. Для этих целей можно использовать *сигнальную метку* – специальное значение, которое сигнализирует об окончании работы. Ниже приводится пример, иллюстрирующий использование значения `None` в качестве сигнальной метки:

```
import multiprocessing

def consumer(input_q):
    while True:
        item = input_q.get()
        if item is None:
            break
        # Обработать элемент
        print(item)          # Заместите эту инструкцию фактической обработкой
    # Завершение
    print("Потребитель завершил работу")

def producer(sequence, output_q):
    for item in sequence:
        # Добавить элемент в очередь
        output_q.put(item)

if __name__ == '__main__':
    q = multiprocessing.Queue()
    # Запустить процесс-потребитель
    cons_p = multiprocessing.Process(target=consumer,args=(q,))
    cons_p.start()

    # Воспроизвести элементы.
    sequence = [1,2,3,4]
    producer(sequence, q)

    # Сообщить о завершении, поместив в очередь сигнальную метку
    q.put(None)

    # Дождаться, пока завершится процесс-потребитель
    cons_p.join()
```

При использовании сигнальных меток, как это показано в примере, следует помнить, что необходимо добавить в очередь по одной сигнальной метке для каждого потребителя. Например, если было запущено три процесса-потребителя, извлекающих элементы из очереди, процесс-поставщик должен добавить в очередь три сигнальные метки, чтобы завершить работу всех потребителей.

Для обмена сообщениями между процессами вместо очередей можно использовать каналы.

`Pipe([`*`duplex`*`])`

Создает канал между процессами и возвращает кортеж (*conn1*, *conn2*), где поля *conn1* и *conn2* являются объектами класса `Connection`, представляющими концы канала. По умолчанию создается двунаправленный канал. Если в аргументе *duplex* передать значение `False`, то объект *conn1* можно будет использовать только для чтения, а объект *conn2* – только для записи. Функция `Pipe()` должна вызываться до создания и запуска объектов класса `Process`, которые будут пользоваться каналом.

Экземпляр *c* класса `Connection`, возвращаемый функцией `Pipe()`, обладает следующими методами и атрибутами:

c`.close()`

Закрывает соединение. Вызывается автоматически, когда объект *c* утилизируется сборщиком мусора.

c`.fileno()`

Возвращает целочисленный дескриптор файла, идентифицирующий соединение.

c`.poll([`*`timeout`*`])`

Возвращает `True` при наличии данных в канале. Аргумент *timeout* определяет предельное время ожидания. При вызове без аргумента метод немедленно возвращает результат. Если в аргументе *timeout* передать значение `None`, метод будет ожидать появления данных неопределенно долго.

c`.recv()`

Извлекает из канала объект, отправленный методом *c*`.send()`. Если с другой стороны соединение было закрыто и в канале нет данных, возбуждает исключение `EOFError`.

c`.recv_bytes([`*`maxlength`*`])`

Принимает строку байтов сообщения, отправленную методом *c*`.send_bytes()`. Аргумент *maxlength* определяет максимальное количество байтов, которые требуется принять. Если входящее сообщение длиннее заданного значения, возбуждается исключение `IOError`, после чего последующие операции чтения из канала становятся невозможными. Если с другой стороны соединение было закрыто и в канале нет данных, возбуждает исключение `EOFError`.

c`.recv_bytes_into(buffer [, offset])`

Принимает строку байтов сообщения и сохраняет ее в объекте *buffer*, который должен поддерживать интерфейс буферов, доступных для записи (такой как объект типа `bytearray` или похожий). Аргумент *offset* определяет смещение в байтах от начала буфера, куда будет записано сообщение. Возвращает количество прочитанных байтов. Если длина сообщения превысит объем доступного пространства в буфере, будет возбуждено исключение `BufferTooShort`.

c`.send(`*`obj`*`)`

Отправляет объект через соединение. Аргумент *obj* может быть любым объектом, совместимым с модулем `pickle`.

c.send_bytes(*buffer* [, *offset* [, *size*]])

Отправляет строку байтов из буфера через соединение. Аргумент *buffer* может быть любым объектом, поддерживающим интерфейс буферов. Аргумент *offset* определяет смещение в байтах от начала буфера, а аргумент *size* – количество байтов, которые требуется отправить. Данные отправляются в виде одного сообщения и могут быть приняты одним вызовом метода *c*.recv_bytes().

Работа с каналами мало чем отличается от работы с очередями. Ниже приводится пример, демонстрирующий решение предыдущей задачи передачи данных между поставщиком и потребителем на основе каналов:

```
import multiprocessing

# Получает элементы из канала.
def consumer(pipe):
    output_p, input_p = pipe
    input_p.close()          # Закрыть конец канала, доступный для записи
    while True:
        try:
            item = output_p.recv()
        except EOFError:
            break
        # Обработать элемент
        print(item)          # Заместите эту инструкцию фактической обработкой
    # Завершение
    print("Потребитель завершил работу")

# Создает элементы и помещает их в канал. Переменная sequence представляет
# итерируемый объект с элементами, которые требуется обработать.
def producer(sequence, input_p):
    for item in sequence:
        # Послать элемент в канал
        input_p.send(item)

if __name__ == '__main__':
     (output_p, input_p) = multiprocessing.Pipe()
    # Запустить процесс-потребитель
    cons_p = multiprocessing.Process(target=consumer,args=((output_p,
                                                            input_p),))
    cons_p.start()

    # Закрыть в поставщике конец канала, доступный для чтения
    output_p.close()

    # Отправить элементы
    sequence = [1,2,3,4]
    producer(sequence, input_p)

    # Сообщить об окончании, закрыв конец канала, доступный для записи
    input_p.close()

    # Дождаться, пока завершится процесс-потребитель
    cons_p.join()
```

Особое внимание должно уделяться корректному управлению концами канала. Если один из концов канала не используется поставщиком или потребителем, его следует закрыть. Именно этим объясняется, почему в примере выше процесс-поставщик закрывает конец канала, доступный для чтения, а процесс-потребитель – конец канала, доступный для записи. Если забыть выполнить одну из этих операций, программа может зависнуть при вызове метода `recv()` в процессе-потребителе. Операционная система ведет подсчет ссылок на каналы, поэтому, чтобы возбудить исключение `EOFError`, канал должен быть закрыт с обоих концов. То есть, если закрыть канал только со стороны поставщика, это не окажет никакого эффекта, пока потребитель не закроет свой конец того же канала.

Каналы могут использоваться как средство обмена сообщениями в двух направлениях. То есть они позволяют писать программы, реализующие модель обмена запрос/ответ, которая обычно применяется во взаимодействиях типа клиент/сервер или в вызовах удаленных процедур. Ниже приводится пример реализации такого типа взаимодействий:

```
import multiprocessing
# Серверный процесс
def adder(pipe):
    server_p, client_p = pipe
    client_p.close()
    while True:
        try:
            x,y = server_p.recv()
        except EOFError:
            break
        result = x + y
        server_p.send(result)
    # Завершение
    print("Сервер завершил работу")

if __name__ == '__main__':
     (server_p, client_p) = multiprocessing.Pipe()
    # Запустить серверный процесс
    adder_p = multiprocessing.Process(target=adder,args=((server_p,
                                                          client_p),))
    adder_p.start()

    # Закрыть серверный канал в клиенте
    server_p.close()

    # Послать серверу несколько запросов
    client_p.send((3,4))
    print(client_p.recv())

    client_p.send(('Hello','World'))
    print(client_p.recv())

    # Конец. Закрыть канал
    client_p.close()

    # Дождаться, пока завершится серверный процесс
    adder_p.join()
```

В этом примере функция adder() запускается как серверный процесс, ожидающий поступления сообщений на своем конце канала. Получив сообщение, сервер обрабатывает его и отправляет результаты обратно в канал. Не забывайте, что методы send() и recv() используют модуль pickle для сериализации и десериализации объектов. В примере сервер получает кортеж (*x*, *y*) и возвращает результат операции *x* + *y*. В более сложных приложениях могут использоваться вызовы удаленных процедур, для чего может потребоваться создать пул процессов, о чем рассказывается ниже.

Пулы процессов

Следующий класс позволяет создать пул процессов, которые могут выполнять различные виды обработки данных. По своей функциональности пул напоминает генератор списков и операции функционального программирования, такие как отображение и снижение размерности.

Pool([*numprocess* [,*initializer* [, *initargs*]]])

Создает пул процессов. В аргументе *numprocess* передается число процессов, которые требуется создать. Если этот аргумент опустить, будет использоваться значение, возвращаемое функцией cpu_count(). В аргументе *initializer* передается вызываемый объект, который будет выполняться при запуске каждого процесса из пула. Аргумент *initargs* – это кортеж параметров, передаваемых объекту *initializer*. По умолчанию аргумент *initializer* принимает значение None.

Экземпляр *p* класса Pool поддерживает следующие операции:

p.apply(*func* [, *args* [, *kwargs*]])

Выполняет функцию *func*(**args*, ***kwargs*) в одном из процессов пула и возвращает результат. Важно отметить, что функция *func* вызывается не во всех, а только в одном процессе из пула. Если потребуется вызвать функцию *func* параллельно, с другим набором аргументов, метод *p*.apply() следует вызвать из другого потока управления или воспользоваться методом *p*.apply_async().

p.apply_async(*func* [, *args* [, *kwargs* [, *callback*]]])

Выполняет функцию *func*(**args*, ***kwargs*) в одном из процессов пула и возвращает результат асинхронно. Результатом вызова этого метода является экземпляр класса AsyncResult, который позднее можно использовать для получения фактического результата. В аргументе *callback* передается вызываемый объект, принимающий единственный аргумент. Как только результат вызова функции *func* будет доступен, он немедленно будет передан объекту *callback*. Объект *callback* не должен выполнять какие-либо операции блокировки, в противном случае он заблокирует прием результатов в других асинхронных операциях.

p.close()

Закрывает пул процессов и предотвращает возможность выполнения последующих операций. Если к этому моменту остались какие-либо незавершенные операции, процессы из пула завершатся только после их выполнения.

p.join()

Ожидает, пока все процессы из пула не завершат работу. Может вызываться только после вызова метода close() или terminate().

p.imap(*func*, *iterable* [, *chunksize*])

Версия функции map(), которая вместо списка с результатами возвращает итератор.

p.imap_unordered(*func*, *iterable* [, *chunksize*]])

То же, что и imap(), за исключением того, что результаты возвращаются в произвольном порядке, в зависимости от очередности получения их от процессов из пула.

p.map(*func*, *iterable* [, *chunksize*])

Применяет вызываемый объект *func* ко всем элементам в объекте *iterable* и возвращает список с результатами. Операция выполняется несколькими процессами параллельно, для чего объект *iterable* разбивается на фрагменты, которые затем передаются процессам из пула. Аргумент *chunksize* определяет количество элементов в каждом фрагменте. При обработке больших объемов данных увеличение значения *chunksize* может привести к улучшению общей производительности.

p.map_async(*func*, *iterable* [, *chunksize* [, *callback*]])

То же, что и map(), за исключением того, что результат возвращается асинхронно. Возвращает экземпляр класса AsyncResult, который позднее может использоваться для получения фактического результата. В аргументе *callback* передается вызываемый объект, принимающий единственный аргумент. Как только результат будет доступен, он немедленно будет передан объекту *callback*.

p.terminate()

Немедленно завершает все процессы из пула, не позволяя им выполнить какие-либо заключительные действия или незавершенные операции. Этот метод вызывается автоматически, когда объект *p* утилизируется сборщиком мусора.

Методы apply_async() и map_async() возвращают экземпляр *a* класса AsyncResult, который обладает следующими методами:

a.get([*timeout*])

Возвращает результат, ожидая, пока он поступит, если это необходимо. В необязательном аргументе *timeout* передается предельное время ожидания. Если результат не поступит в течение указанного промежутка времени, возбуждается исключение multiprocessing.TimeoutError. Если при выполнении удаленной операции возникло исключение, оно повторно будет возбуждено при вызове этого метода.

a.ready()

Возвращает True, если вызов завершился полностью.

a.sucessful()

Возвращает True, если в процессе вызова не возникло исключений. Если этот метод будет вызван до того, как результаты станут доступны, возбуждается исключение AssertionError.

a.wait([*timeout*])

Ожидает, пока результаты станут доступны. В необязательном аргументе *timeout* передается предельное время ожидания.

Следующий пример демонстрирует использование пула процессов для построения словаря, отображающего имена файлов в значения контрольных сумм SHA512 для всех файлов в каталоге:

```
import os
import multiprocessing
import hashlib

# Вы можете изменить значения некоторых параметров
BUFSIZE = 8192                # Размер буфера чтения
POOLSIZE = 2                  # Количество процессов

def compute_digest(filename):
    try:
        f = open(filename,"rb")
    except IOError:
        return None
    digest = hashlib.sha512()
    while True:
        chunk = f.read(BUFSIZE)
        if not chunk: break
        digest.update(chunk)
    f.close()
    return filename, digest.digest()

def build_digest_map(topdir):
    digest_pool = multiprocessing.Pool(POOLSIZE)
    allfiles = (os.path.join(path,name)
                   for path, dirs, files in os.walk(topdir)
                       for name in files)
    digest_map = dict(digest_pool.imap_unordered(compute_digest, allfiles,20))
    digest_pool.close()
    return digest_map

# Проверка. Измените имя каталога на желаемое.
if __name__ == '__main__':
    digest_map = build_digest_map("/Users/beazley/Software/Python-3.0")
    print(len(digest_map))
```

Сначала в этом примере с помощью выражения-генератора определяется последовательность всех путей к файлам в указанном дереве каталогов. Затем эта последовательность делится на фрагменты и частями передается процессам из пула с помощью метода imap_unordered(). Каждый процесс из пула вычисляет контрольные суммы SHA512 для своих файлов с помощью функции compute_digest() и возвращает результаты главному процессу,

который собирает их в словаре. Несмотря на то что этот пример не содержит особых программистских ухищрений, тем не менее на макбуке автора с двухядерным процессором он дает 75-процентный прирост скорости по сравнению с решением, основанным на использовании единственного процесса.

Имейте в виду, что использовать пул процессов имеет смысл только в том случае, если объем вычислений достаточно велик, чтобы компенсировать дополнительные накладные расходы на взаимодействия между процессами. Вообще говоря, нет смысла использовать пул процессов для простых вычислений, таких как сложение пары чисел.

Совместно используемые данные и синхронизация

Обычно процессы полностью изолированы друг от друга и могут обмениваться между собой только с помощью очередей или каналов. Однако существует два объекта, которые могут использоваться для представления разделяемых данных. Внутри эти объекты используют области разделяемой памяти (с помощью модуля `mmap`), обеспечивая доступ к ним из нескольких процессов.

`Value(`*`typecode`*`, `*`arg1`*`, ... `*`argN`*`, `*`lock`*`)`

Создает объект типа `ctypes` в разделяемой памяти. В аргументе *`typecode`* передается либо строка, содержащая код типа из модуля `array` (например, `'i'`, `'d'` и другие), либо объект типа из модуля `ctypes` (например, `ctypes.c_int`, `ctypes.c_double` и другие). Все остальные позиционные аргументы *`arg1`*, *`arg2`*, ..., `argN` передаются конструктору указанного типа. Аргумент *`lock`* может передаваться только как именованный аргумент. Если он имеет значение `True` (по умолчанию), создается новая блокировка для защиты доступа к значению. Если передать в этом аргументе существующую блокировку, например экземпляр класса `Lock` или `RLock`, то для синхронизации доступа будет использоваться эта блокировка. Если допустить, что *`v`* – это экземпляр разделяемого значения, созданного функцией `Value()`, то само значение будет доступно как *`v`*`.value`. Например, обращение к атрибуту *`v`*`.value` вернет значение, а операция присваивания атрибуту *`v`*`.value` изменит значение.

`RawValue(`*`typecode`*`, `*`arg1`*`, ..., `*`argN`*`)`

То же, что и `Value()`, за исключением того, что не использует блокировку.

`Array(`*`typecode`*`, `*`initializer`*`, `*`lock`*`)`

Создает в разделяемой памяти массив объектов `ctypes`. Аргумент *`typecode`* определяет тип элементов массива и имеет тот же смысл, что и в функции `Value()`. В аргументе *`initializer`* можно передать либо целое число, определяющее начальный размер массива, либо последовательность элементов, элементы которой будет использоваться для инициализации массива. Аргумент *`lock`* может передаваться только как именованный аргумент и имеет тот же смысл, что и в функции `Value()`. Если допустить, что *`a`* – это экземпляр разделяемого массива, созданного функцией `Array()`, то к элементам массива можно будет обращаться с использованием стандартных операций индексирования, извлечения среза и итераций, каждая из которых будет синхронизироваться блокировкой. В случае строк байтов объект *`a`* будет

также обладать атрибутом *a*.value, позволяющим обращаться к массиву как к единой строке.

RawArray(*typecode*, *initializer*)

То же, что и Array(), за исключением того, что не использует блокировку. Если потребуется написать программу, которая оперирует одновременно большим количеством элементов массива, для повышения производительности лучше будет использовать этот тип данных и предусмотреть отдельную блокировку для синхронизации (если это необходимо).

В дополнение к разделяемым значениям, которые создаются функциями Value() и Array(), модуль multiprocessing предоставляет разделяемые версии следующих примитивов синхронизации:

Примитив	Описание
Lock	Взаимоисключающая блокировка
Rlock	Реентерабельная взаимоисключающая блокировка (может приобретаться одним и тем же процессом множество раз, не блокируя его)
Semaphore	Семафор
BoundedSemaphore	Ограниченный семафор
Event	Событие
Condition	Переменная состояния

Своим поведением эти объекты имитируют примитивы синхронизации идентичными именами, объявленные в модуле threading. За дополнительной информацией обращайтесь к документации модуля threading.

Следует отметить, что при работе с модулем multiprocessing обычно не приходится использовать блокировки, семафоры и другие подобные механизмы синхронизации, настолько же низкоуровневые, какие используются при работе с потоками. Операции send() и recv() каналов и операции put() и get() очередей уже обеспечивают синхронизацию. Однако в некоторых особых случаях вполне могут найти применение разделяемые значения и блокировки. Ниже приводится пример, в котором для передачи списка чисел с плавающей точкой от одного процесса другому вместо канала используется разделяемый массив:

```
import multiprocessing

class FloatChannel(object):
    def __init__(self, maxsize):
        self.buffer     = multiprocessing.RawArray('d',maxsize)
        self.buffer_len = multiprocessing.Value('i')
        self.empty      = multiprocessing.Semaphore(1)
        self.full       = multiprocessing.Semaphore(0)
    def send(self,values):
        self.empty.acquire()          # Продолжить, только если буфер пуст
        nitems = len(values)
```

```
            self.buffer_len = nitems      # Установить размер буфера
            self.buffer[:nitems] = values # Скопировать значения в буфер
            self.full.release()           # Сообщить, что буфер заполнен
        def recv(self):
            self.full.acquire()           # Продолжить, только если буфер заполнен
            values = self.buffer[:self.buffer_len.value] # Скопировать значение
            self.empty.release()          # Сообщить, что буфер пуст
            return values

    # Проверка производительности. Прием пакета сообщений
    def consume_test(count, ch):
        for i in xrange(count):
            values = ch.recv()

    # Проверка производительности. Передача пакета сообщений
    def produce_test(count, values, ch):
        for i in xrange(count):
            ch.send(values)

    if __name__ == '__main__':
        ch = FloatChannel(100000)
        p = multiprocessing.Process(target=consume_test,
                                    args=(1000,ch))
        p.start()
        values = [float(x) for x in xrange(100000)]
        produce_test(1000, values, ch)
        print("Конец")
        p.join()
```

Подробное изучение этого примера я оставлю вам, в качестве самостоятельного упражнения. Однако замечу, что при проверке производительности на компьютере автора отправка огромного списка чисел с плавающей точкой с помощью объекта `FloatChannel` выполняется примерно на 80 процентов быстрее, чем отправка того же списка с помощью объекта `Pipe` (который вынужден преобразовывать все значения в последовательную форму и обратно с помощью модуля `pickle`).

Управляемые объекты

В отличие от потоков управления, процессы не поддерживают разделяемые объекты. Несмотря на имеющуюся возможность создавать разделяемые значения и массивы, как было показано в предыдущем разделе, этот прием не может использоваться для передачи более сложных объектов языка Python, таких как словари, списки или экземпляры пользовательских классов. Однако модуль `multiprocessing` предоставляет способ совместного использования объектов, при условии, что они будут действовать под управлением так называемого *менеджера*. *Менеджер* – это отдельный дочерний процесс, в котором существует действующий объект и который играет роль сервера. Другие процессы обращаются к разделяемым объектам с помощью специальных промежуточных прокси-объектов, которые действуют как клиенты сервера менеджера.

Самый простой способ создать разделяемый объект заключается в том, чтобы вызвать функцию `Manager()`.

`Manager()`

Создает и запускает отдельный процесс сервера менеджера. Возвращает экземпляр класса `SyncManager`, который определен в подмодуле `multiprocessing.managers`.

Экземпляр *m* класса *SyncManager*, возвращаемый функцией `Manager()`, обладает группой методов, создающих разделяемые объекты и возвращающих прокси-объекты, которые могут использоваться для доступа к разделяемым объектам. Обычно создание менеджера и создание разделяемых объектов с помощью его методов производится перед запуском любых новых процессов. Ниже перечислены доступные методы:

m`.Array(`*typecode*, *sequence*`)`

Создает на стороне сервера разделяемый объект типа `Array` и возвращает прокси-объект для доступа к нему. Описание аргументов можно найти в разделе «Совместно используемые данные и синхронизация».

m`.BoundedSemaphore([`*value*`])`

Создает на стороне сервера разделяемый экземпляр `threading.BoundedSemaphore` и возвращает прокси-объект для доступа к нему.

m`.Condition([`*lock*`])`

Создает на стороне сервера разделяемый экземпляр `threading.Condition` и возвращает прокси-объект для доступа к нему. В аргументе *lock* передается прокси-объект, созданный вызовом метода *m*`.Lock()` или *m*`.Rlock()`.

m`.dict([`*args*`])`

Создает на стороне сервера разделяемый объект типа `dict` и возвращает прокси-объект для доступа к нему. Этот метод принимает те же аргументы, что и встроенная функция `dict()`.

m`.Event()`

Создает на стороне сервера разделяемый экземпляр `threading.Event` и возвращает прокси-объект для доступа к нему.

m`.list([`*sequence*`])`

Создает на стороне сервера разделяемый объект типа `list` и возвращает прокси-объект для доступа к нему. Этот метод принимает те же аргументы, что и встроенная функция `list()`.

m`.Lock()`

Создает на стороне сервера разделяемый экземпляр `threading.Lock` и возвращает прокси-объект для доступа к нему.

m`.Namespace()`

Создает на стороне сервера разделяемый объект пространства имен и возвращает прокси-объект для доступа к нему. Под *пространством имен* в данном случае подразумевается объект, напоминающий модуль на языке Python. Например, если допустить, что *n* – это прокси-объект для доступа к объекту пространства имен, то к его атрибутам можно обращаться с помощью оператора точки (.), например: *n.name = value* или *value = n.name*.

Следует заметить, что имя атрибута *name* в подобных операциях имеет большое значение. Если имя атрибута *name* начинается с алфавитного символа, то его значением является часть разделяемого объекта, находящегося под управлением менеджера, и он будет доступен всем остальным процессам. Если имя атрибута *name* начинается с символа подчеркивания, этот атрибут является частью прокси-объекта и не будет доступен другим процессам.

m.Queue()

Создает на стороне сервера разделяемый экземпляр Queue.Queue и возвращает прокси-объект для доступа к нему.

m.RLock()

Создает на стороне сервера разделяемый экземпляр threading.Rlock и возвращает прокси-объект для доступа к нему.

m.Semaphore([value])

Создает на стороне сервера разделяемый экземпляр threading.Semaphore и возвращает прокси-объект для доступа к нему.

m.Value(*typecode*, *value*)

Создает на стороне сервера разделяемый объект типа Value и возвращает прокси-объект для доступа к нему. Описание аргументов можно найти в разделе «Совместно используемые данные и синхронизация».

Следующий пример демонстрирует, как с помощью менеджера можно создать словарь для совместного использования несколькими процессами.

```
import multiprocessing
import time

# выводит содержимое словаря d всякий раз,
# когда устанавливается переданное событие
def watch(d, evt):
    while True:
        evt.wait()
        print(d)
        evt.clear()

if __name__ == '__main__':
    m = multiprocessing.Manager()
    d = m.dict()        # Создать разделяемый словарь
    evt = m.Event()     # Создать разделяемое событие

    # Запустить процесс, который выводит содержимое словаря
    p = multiprocessing.Process(target=watch,args=(d,evt))
    p.daemon=True
    p.start()

    # Добавить элемент словаря и известить процесс вывода его содержимого
    d['foo'] = 42
    evt.set()
    time.sleep(5)

    # Добавить элемент словаря и известить процесс вывода его содержимого
    d['bar'] = 37
```

```
evt.set()
time.sleep(5)

# Завершить процесс вывода и процесс менеджера
p.terminate()
m.shutdown()
```

Если запустить этот пример, функция `watch()` будет выводить содержимое словаря `d` всякий раз, когда будет генерироваться событие. Главная программа создает разделяемый словарь и событие и манипулирует ими в главном процессе. После запуска примера можно увидеть, как дочерний процесс выводит данные.

Если в программе потребуется использовать разделяемые объекты других типов, например экземпляры пользовательских классов, можно создать собственный объект менеджера. Для этого нужно определить класс, производный от класса `BaseManager`, который объявлен в подмодуле `multiprocessing.managers`.

`managers.BaseManager([`*`address`* `[,` *`authkey`*`]])`

Базовый класс `BaseManager` используется для создания нестандартных классов-менеджеров, управляющих пользовательскими объектами. В необязательном аргументе *address* передается кортеж (*hostname, port*), определяющий сетевой адрес сервера. Если этот аргумент опустить, операционная система просто присвоит адрес, соответствующий свободному номеру порта. В аргументе *authkey* передается строка, которая будет использоваться для аутентификации клиентов при подключении к серверу. Если этот аргумент опустить, будет использоваться значение `current_process().authkey`.

Экземпляр *mgrclass* класса, производного от класса `BaseManager`, может использовать следующий метод класса для создания методов возвращаемого прокси-объекта, обеспечивающего доступ к разделяемым объектам.

mgrclass`.register(`*`typeid`* `[,` *`callable`* `[,` *`proxytype`* `[,` *`exposed`* `[,` *`method_to_typeid`* `[,` *`create_method`*`]]]]])`

Регистрирует новый тип данных в классе менеджера. Аргумент *typeid* — строка, которая используется в качестве имени типа разделяемого объекта. Эта **строка должна быть допустимым идентификатором. В аргументе** *callable* передается вызываемый объект, создающий и возвращающий экземпляр разделяемого объекта. В аргументе *proxytype* передается класс, представляющий реализацию прокси-объектов, которые будут использоваться клиентами. Обычно эти классы создаются автоматически, поэтому часто в этом аргументе передается значение `None`. В аргументе *exposed* передается последовательность имен методов разделяемого объекта, к которым можно будет обращаться с помощью прокси-объекта. Если опустить этот аргумент, будет использоваться значение атрибута *proxytype*`._exposed_`, а если и аргумент *proxytype* не определен, то будет создан список всех общедоступных методов (все методы, имена которых не начинаются с символа подчеркивания (_)). В аргументе *method_to_typeid* передается отображение имен методов в идентификаторы типов возвращаемых ими значений, которое будет использоваться, чтобы определить, какие методы должны воз-

вращать результаты с помощью прокси-объекта. Возвращаемые значения методов, отсутствующих в этом отображении, будут просто копироваться и возвращаться. Если в аргументе *method_to_typeid* передать None, будет использовано значение атрибута *proxytype*._method_to_typeid_, если этот аргумент определен. В аргументе *create_method* передается логический флаг, определяющий, должен ли создаваться метод с именем *typeid* в экземпляре *mgrclass*. По умолчанию принимает значение True.

Чтобы задействовать экземпляр *m* класса менеджера, производного от класса BaseManager, его необходимо запустить вручную. Ниже перечислены атрибуты и методы, имеющие прямое отношение к этому:

m.address

Кортеж (*hostname*, *port*) с адресом, который будет использоваться сервером менеджера.

m.connect()

Устанавливает соединение с удаленным объектом менеджера, адрес которого был указан при вызове конструктора класса BaseManager.

m.serve_forever()

Запускает сервер менеджера в текущем процессе.

m.shutdown()

Останавливает сервер менеджера, запущенный методом *m*.start().

m.start()

Запускает дочерний процесс и затем запускает сервер менеджера в этом процессе.

Следующий пример демонстрирует, как создаются менеджеры, управляющие объектами пользовательских классов:

```
import multiprocessing
from multiprocessing.managers import BaseManager

class A(object):
    def __init__(self,value):
        self.x = value
    def __repr__(self):
        return "A(%s)" % self.x
    def getX(self):
        return self.x
    def setX(self,value):
        self.x = value
    def __iadd__(self,value):
        self.x += value
        return self

class MyManager(BaseManager): pass
MyManager.register("A",A)

if __name__ == '__main__':
    m = MyManager()
    m.start()
```

```
# Создать объект менеджера
a = m.A(37)
...
```

Последняя инструкция в этом примере создает экземпляр класса `A` в процессе сервера менеджера. Переменная *a* в предыдущем примере – это всего лишь прокси-объект для данного экземпляра. Своим поведением прокси-объект напоминает (близко, но не полностью) объект, на который он ссылается. Прежде всего, без труда можно обнаружить, что не все атрибуты и свойства разделяемого объекта доступны непосредственно. Вместо прямого обращения к атрибуту следует использовать метод доступа:

```
>>> a.x
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'AutoProxy[A]' object has no attribute 'x'
(Перевод:
Трассировочная информация (самый последний вызов - самый нижний):
  Файл "<stdin>", строка 1, в <модуль>
AttributeError: Объект 'AutoProxy[A]' не имеет атрибута 'x'
)
>>> a.getX()
37
>>> a.setX(42)
>>>
```

При передаче прокси-объекта функции `repr()` она возвращает строковое представление самого прокси-объекта, тогда как функция `str()` возвращает результат вызова метода `__repr__()` разделяемого объекта. Например:

```
>>> a
<AutoProxy[A] object, typeid 'A' at 0xcef230>
>>> print(a)
A(37)
>>>
```

Специальные методы, а также любые методы, имена которых начинаются с символа подчеркивания (_), недоступны прокси-объектам. Например, попытка вызвать метод `a.__iadd__()` не увенчается успехом:

```
>>> a += 37
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +=: 'AutoProxy[A]' and 'int'
(Перевод:
Трассировочная информация (самый последний вызов - самый нижний):
  Файл "<stdin>", строка 1, в <модуль>
TypeError: неподдерживаемые типы операндов для +=: 'AutoProxy[A]' и 'int'
)
>>> a.__iadd__(37)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'AutoProxy[A]' object has no attribute '__iadd__'
(Перевод:
Трассировочная информация (самый последний вызов - самый нижний):
```

```
  Файл "<stdin>", строка 1, в <модуль>
AttributeError: Объект 'AutoProxy[A]' не имеет атрибута '__iadd__'
)
>>>
```

В более сложных ситуациях можно создать собственный прокси-объект, реализовав в нем более полное управление доступом. Для этого следует определить класс, производный от класса `BaseProxy`, объявленного в модуле `multiprocessing.managers`. В следующем фрагменте демонстрируется, как можно создать собственный прокси-объект для класса `A` из предыдущего примера, который обеспечивает доступ к методу `__iadd__()` и определяет свойство для доступа к атрибуту `x`:

```
from multiprocessing.managers import BaseProxy

class AProxy(BaseProxy):
    # Список всех методов разделяемого объекта, доступных через прокси-объект
    _exposed_ = ['__iadd__','getX','setX']
    # Реализация общедоступного интерфейса прокси-объекта
    def __iadd__(self,value):
        self._callmethod('__iadd__',(value,))
        return self
    @property
    def x(self):
        return self._callmethod('getX',())
    @x.setter
    def x(self,value):
        self._callmethod('setX',(value,))

class MyManager(BaseManager): pass
MyManager.register("A", A, proxytype=AProxy)
```

Экземпляр *proxy* класса, производного от класса `BaseProxy`, обладает следующими методами:

proxy.`_callmethod(`*name* [, *args* [, *kwargs*]]`)`

Вызывает метод *name* разделяемого объекта, на который ссылается прокси-объект. В аргументе *name* передается строка с именем метода, в *args* передается кортеж с позиционными аргументами и в *kwargs* – словарь с именованными аргументами вызываемого метода. Имя *name* метода должно быть явно определено в прокси-объекте. Обычно это делается посредством включения имени в атрибут класса `_exposed_` прокси-объекта.

proxy.`_getvalue()`

Возвращает вызывающему программному коду копию разделяемого объекта, на который ссылается прокси-объект. Если вызов производится из другого процесса, копируемый объект сериализуется, передается вызывающему процессу и затем восстанавливается. Если разделяемый объект не поддерживает возможность сериализации, возбуждается исключение.

Соединения

Программы, использующие модуль `multiprocessing`, имеют возможность обмениваться сообщениями с другими процессами, выполняющимися как на

локальном, так и на удаленном компьютере. Это может быть удобно, когда потребуется взять за основу некоторую программу, предназначенную для работы на единственном компьютере, и добавить в нее возможность выполняться на кластере. В подмодуле `multiprocessing.connection` имеются функции и классы, способные помочь в решении этой задачи:

`connections.Client(`*address* [, *family* [, *authenticate* [, *authkey*]]])

Устанавливает соединение с другим процессом, который уже должен ожидать поступление запросов на соединение по адресу *address*. В аргументе *address* может передаваться кортеж (*hostname*, *port*), представляющий сетевой адрес, строка с именем файла в формате сокетов UNIX или строка вида *r'\\servername\pipe\pipename'*, представляющая именованный канал Windows в удаленной системе *servername* (для обозначения локального компьютера в поле *servername* следует указывать '.').

В аргументе *family* передается строка, обозначающая формат адреса *address*, которая обычно принимает одно из значений 'AF_INET', 'AF_UNIX' или 'AF_PIPE'. В случае его отсутствия значение этого аргумента по умолчанию определяется, исходя из формата аргумента *address*. В аргументе *authentication* передается логический флаг, определяющий, должна ли выполняться аутентификация. В аргументе *authkey* передается строка с ключом аутентификации. **Если опустить этот аргумент, в качестве ключа будет использоваться значение** `current_process().authkey`. Возвращаемым значением этой функции является объект класса `Connection`, который был описан выше, в разделе «Взаимодействия между процессами».

`connections.Listener([`*address*`[, `*family*`[, `*backlog*`[, `*authenticate*`[, `*authkey*`]]]]])`

Возвращает экземпляр класса `Listener`, реализующего сервер, принимающий и обрабатывающий запросы на соединение, выполняемые функцией `Client()`. Аргументы *address*, *family*, *authenticate* и *authkey* имеют тот же смысл, что и в функции `Client()`. В аргументе *backlog* передается целое число, соответствующее значению, которое передается методу `listen()` сокета, если аргумент *address* соответствует сетевому соединению. По умолчанию аргумент *backlog* принимает значение 1. Если аргумент *address* не указан, то используется адрес по умолчанию. Если отсутствуют оба аргумента, *address* и *family*, то выбирается самая быстрая схема взаимодействий в пределах локальной системы.

Экземпляр *s* класса `Listener` поддерживает следующие методы и атрибуты:

s`.accept()`

Принимает запрос на соединение и возвращает объект класса `Connection`. В случае ошибки аутентификации возбуждает исключение `AuthenticationError`.

s`.address`

Адрес, используемый принимающей стороной.

s`.close()`

Закрывает канал или сокет, используемый принимающей стороной.

s.last_accepted

Адрес последнего клиента, с которым было установлено соединение.

Ниже приводится пример программы-сервера, которая ожидает появления запросов на соединение от клиентов и реализует простейшую удаленную операцию (сложение):

```
from multiprocessing.connection import Listener

serv = Listener(('',15000),authkey='12345')
while True:
    conn = serv.accept()
    while True:
        try:
            x,y = conn.recv()
        except EOFError:
            break
        result = x + y
        conn.send(result)
    conn.close()
```

Ниже приводится пример простой программы-клиента, которая соединяется с сервером и отправляет ему несколько сообщений:

```
from multiprocessing.connection import Client
conn = Client(('localhost',15000), authkey="12345")

conn.send((3,4))
r = conn.recv()
print(r)           # Выведет '7'

conn.send(("Hello","World"))
r = conn.recv()
print(r)          # Выведет 'HelloWorld'
conn.close()
```

Различные вспомогательные функции

В модуле также определены следующие вспомогательные функции:

active_children()

Возвращает список объектов класса Process, соответствующих всем дочерним процессам.

cpu_count()

Возвращает количество процессоров в системе, если их число можно определить.

current_process()

Возвращает объект класса Process, соответствующий текущему процессу.

freeze_support()

Вызов этой функции должен включаться в главную программу в виде первой инструкции, если приложение должно поддерживать компиляцию

в двоичные файлы с помощью различных инструментов создания пакетов, таких как `py2exe`. Это необходимо для предотвращения появления ошибок времени выполнения, связанных с запуском дочерних процессов из приложений, скомпилированных в двоичный формат.

`get_logger()`

Возвращает объект регистратора для модуля `multiprocessing`; создает его, если он еще не был создан. Возвращаемый объект регистратора не передает журналируемые сообщения корневому регистратору, имеет уровень `logging.NOTSET` и выводит все сообщения в поток стандартного вывода сообщений об ошибках.

`set_executable(`*`executable`*`)`

Устанавливает путь к выполняемому файлу интерпретатора Python, который будет использоваться для запуска дочерних процессов. Эта функция определена только в версии модуля для Windows.

Общие советы по использованию модуля multiprocessing

Модуль `multiprocessing` является одним из самых сложных и мощных модулей в стандартной библиотеке языка Python. Ниже приводятся несколько общих советов, которые помогут вам избежать головной боли при работе с ним:

- Внимательно прочитайте электронную документацию, прежде чем приступать к созданию крупного приложения. Хотя в этом разделе были рассмотрены самые важные основы, тем не менее в официальной документации описывается множество более коварных проблем, с которыми можно столкнуться.
- Обязательно убедитесь, что все типы данных, которые участвуют в обмене между процессами, совместимы с модулем `pickle`.
- Старайтесь не использовать механизмы совместного использования данных и учитесь пользоваться механизмами передачи сообщений и очередями. При использовании механизмов обмена сообщениями вам не придется беспокоиться о синхронизации, блокировках и других проблемах. Кроме того, такой подход обладает лучшей масштабируемостью при увеличении числа процессов.
- Не пользуйтесь глобальными переменными в функциях, которые предназначены для работы в отдельных процессах. Вместо этого лучше использовать явную передачу параметров.
- Старайтесь не смешивать поддержку механизма потоков и процессов в одной программе, если только вы не стремитесь существенно повысить защищенность своих разработок от постороннего взгляда (или понизить – в зависимости от того, кто будет просматривать ваш программный код).
- Особое внимание обращайте на то, как завершаются процессы. Как правило, предпочтительнее явно закрывать процессы и предусматри-

вать четко оформленную процедуру их завершения, а не полагаться на механизм сборки мусора или принудительное завершение дочерних процессов, с использованием операции terminate().

- Использование менеджеров и прокси-объектов тесно связано с различными понятиями распределенных вычислений (такими как распределенные объекты). Хорошая книга о распределенных вычислениях может оказаться очень полезным справочником.
- Модуль multiprocessing возник из сторонней библиотеки, известной как pyprocessing. Поиск советов по использованию этой библиотеки и ее описание может принести неплохие результаты.
- Несмотря на то что этот модуль можно использовать в операционной системе Windows, тем не менее вам следует тщательно проработать официальную документацию, чтобы понять самые тонкие особенности. Например, модуль multiprocessing предусматривает реализацию собственного клона функции fork() из UNIX для запуска новых процессов в Windows. Эта функция создает копию окружения родительского процесса и отправляет ее дочернему процессу с помощью канала. Вообще говоря, этот модуль больше подходит для использования в UNIX.
- И самое главное – старайтесь сделать реализацию максимально простой.

Модуль threading

Модуль threading содержит определение класса Thread и реализацию различных механизмов синхронизации, используемых в многопоточных программах.

Объекты класса Thread

Класс Thread используется для представления отдельного потока управления. Новый поток можно создать вызовом конструктора:

```
Thread(group=None, target=None, name=None, args=(), kwargs={})
```

Создает новый экземпляр класса Thread. Аргумент *group* всегда получает значение None и зарезервирован для использования в будущем. В аргументе *target* передается вызываемый объект, который вызывается методом run() при запуске потока. По умолчанию получает значение None, которое означает, что ничего вызываться не будет. Аргумент *name* определяет имя потока. По умолчанию генерируется уникальное имя вида "Thread-N". В аргументе *args* передается кортеж позиционных аргументов для функции *target*, а в аргументе *kwargs* – словарь именованных аргументов для *target*.

Экземпляр *t* класса Thread поддерживает следующие методы и атрибуты:

```
t.start()
```

Запускает поток вызовом метода run() в отдельном потоке управления. Этот метод может вызываться только один раз.

t.run()

Этот метод вызывается при запуске потока. По умолчанию вызывает функцию *target*, которая была передана конструктору. При желании можно создать свой класс, производный от класса Thread, и определить в нем собственную реализацию метода run().

t.join([*timeout*])

Ожидает завершения потока или истечения указанного интервала времени. Аргумент *timeout* определяет максимальный период ожидания в секундах, в виде числа с плавающей точкой. Поток не может присоединяться к самому себе и будет ошибкой пытаться присоединиться к потоку до того, как он будет запущен.

t.is_alive()

Возвращает True, если поток *t* продолжает работу, и False — в противном случае. Поток считается действующим от момента вызова метода start() до момента, когда завершится метод run(). В старых программах можно встретить вызов метода с именем *t*.isAlive(), которое является синонимом этого метода.

t.name

Имя потока. Этот атрибут используется только для идентификации и может принимать любые значения (желательно осмысленные, что может упростить отладку). В старых программах можно встретить вызовы методов с именами *t*.getName() и *t*.setName(*name*), которые используются для манипулирования именем потока.

t.ident

Целочисленный идентификатор потока. Если поток еще не был запущен, этот атрибут содержит значение None.

t.daemon

Логический флаг, указывающий, будет ли поток демоническим. Значение этого атрибута должно устанавливаться до вызова метода start(). По умолчанию он получает значение, унаследованное от потока, создавшего его. Программа на языке Python завершается, когда не осталось ни одного активного, не демонического потока управления. Любая программа имеет главный поток, представляющий первоначальный поток управления, который не является демоническим. В старых программах можно встретить вызовы методов с именами *t*.setDaemon(*flag*) и *t*.isDaemon(), которые используются для манипулирования этим значением.

Ниже приводится пример, демонстрирующий, как создавать и запускать функции (или другие вызываемые объекты) в отдельных потоках управления:

```
import threading
import time

def clock(interval):
    while True:
```

```
        print("Текущее время: %s" % time.ctime())
        time.sleep(interval)

t = threading.Thread(target=clock, args=(15,))
t.daemon = True
t.start()
```

Следующий пример демонстрирует, как определить тот же поток в виде класса:

```
import threading
import time

class ClockThread(threading.Thread):
    def __init__(self,interval):
        threading.Thread.__init__(self)
        self.daemon = True
        self.interval = interval
    def run(self):
        while True:
            print("Текущее время: %s" % time.ctime())
            time.sleep(self.interval)

t = ClockProcess(15)
t.start()
```

Когда объявляется собственный класс потока, в котором переопределяется метод `__init__()`, чрезвычайно важно не забыть вызвать конструктор базового класса `Thread.__init__()`, как показано в примере. Если **этого не сделать**, вы столкнетесь с неприятной ошибкой. Кроме того, ошибкой будет пытаться переопределить какие-либо другие методы класса `Thread`, кроме методов `run()` и `__init__()`.

Настройка атрибута `daemon` в этих примерах является характерной операцией при работе с потоками, которые выполняют бесконечный цикл. Обычно интерпретатор **Python ожидает завершения всех потоков, прежде чем завершиться** самому. Однако такое поведение часто бывает нежелательным, когда имеются никогда не завершающиеся фоновые потоки. Значение `True` в атрибуте `daemon` позволяет интерпретатору завершиться сразу же после выхода из главной программы. В этом случае демонические потоки просто уничтожаются.

Объекты класса Timer

Объекты класса `Timer` используются для вызова функций через определенное время.

`Timer(`*`interval`*`, `*`func`*` [, `*`args`*` [, `*`kwargs`*`]])`

Создает объект таймера, который вызывает функцию *`func`* через *`interval`* секунд. В аргументах *`args`* и *`kwargs`* передаются позиционные и именованные аргументы для функции *`func`*. Таймер не запускается, пока не будет вызван метод `start()`.

Экземпляр *t* класса `Timer` обладает следующими методами:

t.`start()`

Запускает таймер. Функция *func*, переданная конструктору `Timer()`, будет вызвана спустя указанное количество секунд, после вызова этого метода.

t.`cancel()`

Останавливает таймер, если функция еще не была вызвана.

Объекты класса Lock

Простейшая блокировка (или *взаимоисключающая блокировка*) – это механизм синхронизации, имеющий два состояния – «закрыто» и «открыто». Для изменения состояния блокировки используются методы `acquire()` и `release()`. Если блокировка находится в состоянии «закрыто», любая попытка приобрести ее будет заблокирована до момента, пока она не будет освобождена. Если сразу несколько потоков управления пытаются приобрести блокировку, только один из них сможет продолжить работу, когда блокировка будет освобождена. Порядок, в каком потоки смогут продолжить работу, заранее не определен.

Новый экземпляр класса `Lock` создается с помощью конструктора:

`Lock()`

Создает новый экземпляр блокировки, которая изначально находится в состоянии «открыто».

Экземпляр *lock* класса `Lock` поддерживает следующие методы:

lock.`acquire([`*blocking* `])`

Приобретает блокировку. Если блокировка находится в состоянии «закрыто», этот метод приостанавливает работу потока, пока блокировка не будет освобождена. Если в аргументе *blocking* передать значение `False`, метод тут же вернет значение `False`, если блокировка не может быть приобретена, и `True` – если блокировку удалось приобрести.

lock.`release()`

Освобождает блокировку. Будет ошибкой пытаться вызвать этот метод, когда блокировка находится в состоянии «открыто» или из другого потока, не из того, где вызывался метод `acquire()`.

Объекты класса RLock

Реентерабельная блокировка – это механизм синхронизации, напоминающий обычную блокировку, но которая в одном и том же потоке может быть приобретена множество раз. Эта особенность позволяет потоку, владеющему блокировкой, выполнять вложенные операции `acquire()` и `release()`. В подобных ситуациях только самый внешний вызов метода `release()` действительно переведет блокировку в состояние «открыто».

Новый экземпляр класса `RLock` создается с помощью конструктора:

`RLock()`

Создает новый экземпляр реентерабельной блокировки.

Экземпляр *rlock* класса `RLock` поддерживает следующие методы:

rlock`.acquire([`*blocking* `])`

Приобретает блокировку. В случае необходимости дальнейшая работа потока приостанавливается, пока блокировка не будет освобождена. Если перед вызовом метода блокировкой не владел ни один поток, она запирается, а уровень рекурсии блокировки устанавливается равным 1. Если вызывающий поток уже владеет блокировкой, уровень рекурсии увеличивается на единицу и метод тут же возвращает управление.

`rlock.release()`

Уменьшает уровень рекурсии. Если значение уровня рекурсии достигло нуля, блокировка переводится в состояние «открыто». В противном случае блокировка остается в состоянии «закрыто». Эта функция должна вызываться только из потока, который владеет блокировкой.

Семафоры и ограниченные семафоры

Семафор – это механизм синхронизации, основанный на счетчике, который уменьшается при каждом вызове метода `acquire()` и увеличивается при каждом вызове метода `release()`. Если счетчик семафора достигает нуля, метод `acquire()` приостанавливает работу потока, пока какой-либо другой поток не вызовет метод `release()`.

`Semaphore([`*value*`])`

Создает новый семафор. Аргумент *value* определяет начальное значение счетчика. При вызове без аргументов счетчик получает значение `1`.

Экземпляр *s* класса `Semaphore` поддерживает следующие методы:

s`.acquire([`*blocking*`])`

Приобретает семафор. Если внутренний счетчик имеет значение больше нуля, этот метод уменьшает его на 1 и тут же возвращает управление. Если значение счетчика равно нулю, этот метод приостанавливает работу потока, пока другой поток не вызовет метод `release()`. Аргумент *blocking* имеет тот же смысл, что и в методе `acquire()` экземпляров классов `Lock` и `RLock`.

s`.release()`

Увеличивает внутренний счетчик семафора на 1. Если перед вызовом метода счетчик был равен нулю и имеется другой поток, ожидающий освобождения семафора, этот поток возобновляет работу. Если сразу несколько потоков управления пытаются приобрести семафор, только в одном из них метод `acquire()` вернет управление. Порядок, в каком потоки смогут продолжить работу, заранее не определен.

`BoundedSemaphore([`*value*`])`

Создает новый семафор. Аргумент *value* определяет начальное значение счетчика. При вызове без аргументов счетчик получает значение `1`. Огра-

ниченный семафор BoundedSemaphore действует точно так же, как и обычный семафор Semaphore, за исключением того, что количество вызовов метода release() не может превышать количество вызовов метода acquire().

Тонкое отличие семафоров от взаимоисключающих блокировок состоит в том, что семафоры могут использоваться в качестве сигналов. Например, методы acquire() и release() могут вызываться из разных потоков управления и обеспечивать взаимодействие между потоками поставщика и потребителя.

```
produced = threading.Semaphore(0)
consumed = threading.Semaphore(1)

def producer():
    while True:
        consumed.acquire()
        produce_item()
        produced.release()

def consumer():
    while True:
        produced.acquire()
        item = get_item()
        consumed.release()
```

Такой способ обмена сигналами, как показан в этом примере, часто реализуется с помощью переменных состояния, о которых рассказывается ниже.

События

События используются для организации взаимодействия потоков. Один поток «посылает» событие, а один или более других потоков ожидают его. Экземпляр класса Event обладает внутренним флагом, который можно установить методом set() и сбросить методом clear(). Метод wait() приостанавливает работу потока, пока флаг не будет установлен.

`Event()`

Создает новый экземпляр класса Event со сброшенным внутренним флагом.

Экземпляр *e* класса Event поддерживает следующие методы:

e`.is_set()`

Возвращает True, если внутренний флаг установлен. В старых программах этот метод вызывается под именем isSet().

e`.set()`

Устанавливает внутренний флаг. После этого все потоки, ожидавшие, пока флаг будет установлен, продолжат свою работу.

e`.clear()`

Сбрасывает внутренний флаг.

e`.wait([`*timeout*`])`

Приостанавливает работу потока, пока не будет установлен внутренний флаг. Если флаг уже был установлен, возвращает управление немедленно.

В противном случае работа потока приостанавливается, пока другой поток не установит флаг вызовом метод `set()` или пока не истечет интервал времени *timeout*. В аргументе *timeout* передается число с плавающей точкой, определяющее предельное время ожидания в секундах.

Экземпляры класса `Event` могут использоваться как средство извещения других потоков, но они не должны использоваться для реализации обмена извещениями, который обычен в схемах взаимодействий поставщик/потребитель. Например, избегайте такого способа использования событий:

```
evt = Event()

def producer():
    while True:
        # создать элемент
        ...
        evt.signal()

def consumer():
    while True:
        # Дождаться появления элемента
        evt.wait()
        # Обработать элемент
        ...
        # Сбросить событие и ждать появления следующего элемента
        evt.clear()
```

Такая реализация работает неустойчиво, потому что поставщик может воспроизвести новый элемент в промежутке между вызовами методов `evt.wait()` и `evt.clear()`. В этой ситуации, сбросив событие, потребитель не сможет обнаружить новый элемент, пока поставщик не создаст еще один. В лучшем случае программа будет испытывать небольшие отклонения, выражающиеся в непредсказуемых задержках обработки новых элементов, а в худшем она может зависнуть, потеряв событие. Для решения подобных проблем лучше использовать переменные состояния.

Переменные состояния

Переменная состояния (condition variable) – это механизм синхронизации, надстроенный на уже имеющейся блокировке, который используется потоками, когда требуется дождаться наступления определенного состояния или появления события. Переменные состояния обычно используются в схемах поставщик-потребитель, когда один поток производит данные, а другой обрабатывает их. Новый экземпляр класса `Condition` создается с помощью конструктора:

`Condition([`*`lock`*`])`

Создает новую переменную состояния. В необязательном аргументе *lock* передается экземпляр класса `Lock` или `RLock`. При вызове без аргумента для использования совместно с переменной состояния создается новый экземпляр класса `RLock`.

Экземпляр *cv* класса `Condition` поддерживает следующие методы:

cv.acquire(**args*)

Приобретает блокировку, связанную с переменной состояния. Этот метод вызывает метод acquire(**args*) блокировки и возвращает результат.

cv.release()

Освобождает блокировку, связанную с переменной состояния. Этот метод вызывает метод release() блокировки.

cv.wait([*timeout*])

Ожидает, пока не будет получено извещение или пока не истечет время ожидания. Этот метод должен вызываться только после того, как вызывающий поток приобретет блокировку. При вызове метода блокировка освобождается, а поток приостанавливается, пока другим потоком не будет вызван метод notify() или notifyAll() переменной состояния. После возобновления метод тут же повторно приобретает блокировку и возвращает управление вызывающему потоку. В аргументе *timeout* передается число с плавающей точкой, определяющее предельное время ожидания в секундах. По истечении указанного интервала времени блокировка снова приобретается, и поток возобновляет работу.

cv.notify([*n*])

Возобновляет работу одного или более потоков, ожидающих изменения данной переменной состояния. Этот метод должен вызываться только после того, как поток приобретет блокировку, и ничего не делает, если отсутствуют потоки, ожидающие изменения этой переменной состояния. Аргумент *n* определяет количество потоков, которые смогут возобновить работу, и по умолчанию получает значение 1. Метод wait() не возвращает управление после возобновления потока, пока не сможет повторно приобрести блокировку.

cv.notify_all()

Возобновляет работу всех потоков, ожидающих изменения переменной состояния. В старых программах этот метод вызывается под именем notifyAll().

Ниже приводится пример использования переменной состояния, который можно использовать как заготовку:

```
cv = threading.Condition()
def producer():
    while True:
        cv.acquire()
        produce_item()
        cv.notify()
        cv.release()

def consumer():
    while True:
        cv.acquire()
        while not item_is_available():
            cv.wait()      # Ожидать появления нового элемента
```

```
        cv.release()
        consume_item()
```

Тонкость в использовании переменных состояния заключается в том, что при наличии нескольких потоков, ожидающих изменения одной и той же переменной состояния, метод `notify()` может возобновить работу одного или более из них (конкретное поведение часто зависит от операционной системы). Вследствие этого существует вероятность, что после возобновления работы поток обнаружит, что интересующее его состояние уже отсутствует. Это объясняет, например, почему в функции `consumer()` используется цикл `while`. Если поток возобновил работу, но элемент уже был обработан другим потоком, он просто опять переходит к ожиданию следующего извещения.

Работа с блокировками

При работе с любыми механизмами синхронизации, такими как `Lock`, `RLock` или `Semaphore`, следует быть очень внимательными. Ошибки в управлении блокировками часто приводят к взаимоблокировкам потоков и к состоянию гонки за ресурсами. Программный код, использующий блокировки, должен гарантировать их освобождение даже в случае появления исключений. Ниже приводится типичный пример такого программного кода:

```
try:
    lock.acquire()
    # критический раздел
    инструкции
    ...
finally:
    lock.release()
```

Кроме того, все блокировки поддерживают протокол менеджера контекста, что позволяет писать более ясный код:

```
with lock:
    # критический раздел
    инструкции
    ...
```

В этом примере блокировка автоматически приобретается инструкцией `with` и освобождается, когда поток управления выходит за пределы контекста.

Также следует избегать писать программный код, который в любой момент времени обладал бы более чем одной блокировкой одновременно. Например:

```
with lock_A:
    # критический раздел A
    инструкции
    ...
with lock_B:
    # критический раздел B
    инструкции
    ...
```

Обычно такой код является отличным источником непонятных взаимоблокировок в программе. Несмотря на существование стратегий, позволяющих избегать взаимоблокировок (например, иерархические блокировки), лучше все-таки полностью отказаться от манеры писать такой код.

Приостановка и завершение потока

Потоки не имеют методов для принудительного их завершения или приостановки. Отсутствие этих методов не является недоработкой, а обусловлено сложностями, свойственными процессу разработки многопоточных программ. Например, если поток владеет блокировкой, то принудительное его завершение или приостановка может вызвать зависание всего приложения. Кроме того, обычно нет никакой возможности просто взять и «освободить все блокировки» по завершении, потому что при сложной процедуре синхронизации потоков часто бывает необходимо точно соблюсти последовательность операций приобретения и освобождения блокировок.

Если в программе потребуется возможность завершения или приостановки потока, ее придется реализовать самостоятельно. Обычно для этого поток проверяет в цикле свое состояние и определяет момент, когда он должен завершиться. Например:

```
class StoppableThread(threading.Thread):
    def __init__(self):
        threading.Thread.__init__()
        self._terminate    = False
        self._suspend_lock = threading.Lock()
    def terminate(self):
        self._terminate = True
    def suspend(self):
        self._suspend_lock.acquire()
    def resume(self):
        self._suspend_lock.release()
    def run(self):
        while True:
            if self._terminate:
                break
            self._suspend_lock.acquire()
            self._suspend_lock.release()
            инструкции
            ...
```

Имейте в виду, что чтобы обеспечить надежную работу при таком подходе, поток не должен выполнять какие-либо операции ввода-вывода, которые могут быть заблокированы. Например, если поток приостанавливается в ожидании поступления новых данных, он не может быть завершен, пока не завершит эту операцию. По этой причине желательно ограничивать ожидание в операциях ввода-вывода некоторым интервалом времени, использовать неблокирующие версии операций ввода-вывода и использовать другие дополнительные возможности, чтобы гарантировать, что проверка необходимости завершения будет выполняться достаточно часто.

Вспомогательные функции

Для работы с потоками управления имеются следующие вспомогательные функции:

`active_count()`

Возвращает текущее количество активных объектов класса `Thread`.

`current_thread()`

Возвращает объект класса `Thread`, соответствующий вызывающему потоку управления.

`enumerate()`

Возвращает список всех активных объектов класса `Thread`.

`local()`

Возвращает объект `local`, который служит хранилищем локальных для потока данных. Для каждого потока этот объект гарантированно будет уникальным.

`setprofile(`*`func`*`)`

Устанавливает функцию, которая будет использоваться для профилирования всех создаваемых потоков. Функция *`func`* будет передаваться функции `sys.setprofile()` перед запуском каждого потока.

`settrace(`*`func`*`)`

Устанавливает функцию, которая будет использоваться для трассировки всех создаваемых потоков. Функция *`func`* будет передаваться функции `sys.settrace()` перед запуском каждого потока.

`stack_size([`*`size`*`])`

Возвращает размер стека, который будет использоваться при создании новых потоков. Если в необязательном аргументе *`size`* функции передается целое число, оно будет определять размер стека для вновь создаваемых потоков. Для обеспечения переносимости программы в аргументе *`size`* следует передавать значения от 32 768 (32 Кбайта) и выше, кратные 4096 (4 Кбайта). Если эта операция не поддерживается системой, возбуждается исключение `ThreadError`.

Глобальная блокировка интерпретатора

Интерпретатор Python выполняется под защитой блокировки, которая позволяет выполняться только одному потоку управления в каждый конкретный момент времени, даже если в системе имеется несколько процессоров. Это обстоятельство существенно ограничивает выгоды, которые могло бы принести использование потоков в программах, выполняющих массивные вычисления. Фактически использование потоков в подобных программах часто приводит к существенному ухудшению производительности по сравнению с однопоточными программами, выполняющими ту же работу. По этой причине потоки управления должны использоваться только в программах, основной задачей которых является выполнение операций

ввода-вывода, таких как сетевые серверы. Для решения задач, связанных с массивными вычислениями, лучше использовать модули расширений на языке C или задействовать модуль `multiprocessing`. Расширения на языке C имеют возможность освободить блокировку интерпретатора и выполняться параллельно, при условии, что они никак не будут взаимодействовать с интерпретатором после освобождения блокировки. Модуль `multiprocessing` позволяет переложить работу на независимые дочерние процессы, которые не ограничиваются этой блокировкой.

Разработка многопоточных программ

Несмотря на имеющуюся возможность писать на языке Python традиционные многопоточные программы, используя различные комбинации блокировок и других механизмов синхронизации, существует еще один стиль программирования, который обладает преимуществами перед всеми остальными, – попытаться организовать многопоточную программу как коллекцию независимых задач, взаимодействующих между собой с помощью очередей сообщений. Об этом рассказывается в следующем разделе (модуль `queue`).

Модуль queue (Queue)

Модуль `queue` (в Python 2 он называется `Queue`) реализует различные типы очередей, поддерживающие возможность доступа из множества потоков и обеспечивающие сохранность информации при обмене данными между несколькими потоками управления.

Модуль `queue` определяет три различных класса очередей:

`Queue([`*`maxsize`*`])`

Создает очередь типа **FIFO (first-in first-out – первым пришел, первым вышел)**. Аргумент *`maxsize`* определяет максимальное количество элементов, которое может поместиться в очередь. При вызове без аргумента или когда значение *`maxsize`* равно 0, размер очереди не ограничивается.

`LifoQueue([`*`maxsize`*`])`

Создает очередь типа LIFO (last-in, first-out – последним пришел, первым вышел), которая также известна, как *стек*.

`PriorityQueue([`*`maxsize`*`])`

Создает очередь с поддержкой приоритетов, в которой все элементы упорядочиваются по приоритетам, от низшего к высшему. Элементами очереди этого типа могут быть только кортежи вида (*`priority, data`*), где поле *`priority`* является числом.

Экземпляр *`q`* любого из классов очередей обладает следующими методами:

`q``.qsize()`

Возвращает примерный размер очереди. Так как другие потоки могут добавлять и извлекать элементы, результат вызова этой функции не может считаться надежным.

q.empty()

Возвращает True, если в момент вызова очередь была пустой, и False — в противном случае.

q.full()

Возвращает True, если в момент вызова очередь была полной, и False — в противном случае.

q.put(*item* [, *block* [, *timeout*]])

Добавляет элемент *item* в очередь. Если необязательный аргумент *block* имеет значение True (по умолчанию), в случае отсутствия свободного пространства в очереди вызывающий поток будет приостановлен. Иначе (когда в аргументе *block* передается значение False) в случае отсутствия свободного пространства в очереди будет возбуждено исключение Full. Аргумент timeout определяет предельное время ожидания в секундах. По истечении времени ожидания будет возбуждено исключение Full.

q.put_nowait(*item*)

Соответствует вызову *q*.put(*item*, False).

q.get([*block* [, *timeout*]])

Удаляет и возвращает элемент из очереди. Если необязательный аргумент *block* имеет значение True (по умолчанию), в случае отсутствия элементов в очереди вызывающий поток будет приостановлен. Иначе (когда в аргументе *block* передается значение False) в случае отсутствия элементов будет возбуждено исключение Empty. Аргумент timeout определяет предельное время ожидания в секундах. По истечении времени ожидания будет возбуждено исключение Empty.

q.get_nowait()

Соответствует вызову *q*.get(0).

q.task_done()

Используется потребителем, чтобы сообщить, что элемент очереди был обработан. Если используется, этот метод должен вызываться один раз для каждого элемента, удаленного из очереди.

q.join()

Приостанавливает поток, пока не будут удалены и обработаны все элементы очереди. Возвращает управление только после того, как для каждого элемента очереди будет вызван метод *q*.task_done().

Пример использования очереди в потоках

Разработку многопоточных программ часто можно упростить за счет использования очередей. Например, вместо того, чтобы использовать разделяемые данные, доступ к которым необходимо осуществлять под защитой блокировки, потоки могут обмениваться информацией с помощью очередей. В этой модели поток, занимающийся обработкой данных, обычно играет роль потребителя. Ниже приводится пример, иллюстрирующий эту концепцию:

```
import threading
from queue import Queue # Use from Queue on Python 2

class WorkerThread(threading.Thread):
    def __init__(self,*args,**kwargs):
        threading.Thread.__init__(self,*args,**kwargs)
        self.input_queue = Queue()
    def send(self,item):
        self.input_queue.put(item)
    def close(self):
        self.input_queue.put(None)
        self.input_queue.join()
    def run(self):
        while True:
            item = self.input_queue.get()
            if item is None:
                break
            # Обработать элемент
            # (замените инструкцию print какими-нибудь полезными операциями)
            print(item)
            self.input_queue.task_done()
        # Конец. Сообщить, что сигнальная метка была принята, и выйти
        self.input_queue.task_done()
        return

# Пример использования
w = WorkerThread()
w.start()
w.send("hello")    # Отправить элемент на обработку (с помощью очереди)
w.send("world")
w.close()
```

Этот класс проектировался очень тщательно. Во-первых, можно заметить, что программный интерфейс этого класса представляет собой подмножество методов объектов класса `Connection`, которые создаются каналами в модуле `multiprocessing`. Это обеспечивает возможность дальнейшего расширения. Например, позднее обрабатывающий поток может быть вынесен в отдельный процесс без переделки программного кода, который посылает данные этому потоку.

Во-вторых, программный интерфейс предусматривает возможность завершения потока. Метод `close()` помещает в очередь сигнальную метку, которая в свою очередь вызывает завершение потока.

Наконец, программный интерфейс в значительной степени напоминает интерфейс сопрограмм. Если для выполнения операций не требуется прибегать к блокировкам, метод `run()` можно будет реализовать как сопрограмму и вообще обойтись без потоков. Этот последний способ может обеспечить более высокую производительность, потому что в нем отсутствует необходимость переключения контекста потоков.

Сопрограммы и микропотоки

В некоторых типах приложений вполне возможно реализовать кооперативную многозадачность в пространстве пользователя, основанную на использовании диспетчера задач и набора генераторов или сопрограмм. Иногда их называют *микропотоками*, однако могут использоваться и другие термины – иногда их называют *тасклетами* (*tasklets*), *зелеными потоками*[1] (*green threads*), *гринлетами* (*greenlets*) и так далее. Обычно этот прием используется в программах, где необходимо управлять большими коллекциями открытых файлов или сокетов. В качестве примера можно привести сетевой сервер, который одновременно должен управлять тысячами соединений с клиентами. Вместо того чтобы создавать тысячи потоков управления, можно использовать асинхронные операции ввода-вывода или операции опроса (модуль `select`) в комбинации с диспетчером задач, который обрабатывает события ввода-вывода.

В основе этой методики лежит тот факт, что инструкция `yield`, используемая в функциях-генераторах и сопрограммах, приостанавливает работу функции, пока не будет вызван метод `next()` или `send()`. Это обеспечивает возможность реализации кооперативной многозадачности между множеством функций-генераторов на основе использования цикла обработки событий. Эту идею иллюстрирует следующий пример:

```
def foo():
    for n in xrange(5):
        print("Я - foo %d" % n)
        yield

def bar():
    for n in xrange(10):
        print("Я - bar %d" % n)
        yield

def spam():
    for n in xrange(7):
        print("Я - spam %d" % n)
        yield

# Создать и заполнить очередь задач
from collections import deque
taskqueue = deque()
taskqueue.append(foo())      # Добавить несколько задач (генераторов)
taskqueue.append(bar())
taskqueue.append(spam())

# Запустить все задачи
while taskqueue:
    # Перейти к следующей задаче
    task = taskqueue.pop()
    try:
```

[1] Зеленые потоки и гринлеты иногда называют «легковесными потоками». – *Прим. перев.*

```
            # Выполнить до следующей инструкции yield и опять поставить в очередь
            next(task)
            taskqueue.appendleft(task)
        except StopIteration:
            # Задача завершилась
            pass
```

Этот прием едва ли можно рекомендовать для использования в программах, выполняющих массивные вычисления. Он в большей степени подходит для распределения работы между задачами, выполняющими операции ввода-вывода, опрос или обработку событий. Более сложный пример использования этого приема можно найти в разделе с описанием модуля `select`, в главе 21 «Работа с сетью и сокеты».

21

Работа с сетью и сокеты

В этой главе описываются модули, используемые для низкоуровневой реализации сетевых серверов и клиентов. Стандартная библиотека языка Python обеспечивает широкую поддержку сетевых операций, начиная от операций с сокетами и заканчивая функциями для работы с прикладными протоколами высокого уровня, такими как HTTP. Для начала будет дано краткое (действительно, очень краткое) введение в разработку сетевых приложений. За дополнительной информацией читателям рекомендуется обращаться к специализированным книгам, таким как «UNIX Network Programming, Volume 1: Networking APIs: Sockets and XTI» У. Ричарда Стивенса (W. Richard Stevens) (издательство Prentice Hall, 1997, ISBN 0-13-490012-X).[1] Модули, предназначенные для работы с протоколами прикладного уровня, описываются в главе 22 «Разработка интернет-приложений».

Основы разработки сетевых приложений

Модули, входящие в стандартную библиотеку Python и предназначенные для разработки сетевых приложений, главным образом поддерживают два протокола Интернета: TCP и UDP. *Протокол TCP* – это надежный протокол с созданием логического соединения, используемый для создания между компьютерами двустороннего канала обмена данными. *Протокол UDP* – это низкоуровневый протокол, обеспечивающий возможность обмена пакетами, с помощью которого компьютеры могут отправлять и получать информацию в виде отдельных пакетов, без создания логического соединения. В отличие от TCP, взаимодействия по протоколу UDP не отличаются надежностью, что усложняет управление ими в приложениях, в которых необходимо гарантировать надежность обмена информацией. По этой причине большинство интернет-приложений используют протокол TCP.

---

[1] Стивенс У. «UNIX. Разработка сетевых приложений». – Пер. с англ. – СПб.: Питер, 2003.

Работа с обоими протоколами осуществляется с помощью программной абстракции, известной как сокет. *Сокет* – это объект, напоминающий файл, позволяющий программе принимать входящие соединения, устанавливать исходящие соединения, а также отправлять и принимать данные. Прежде чем два компьютера смогут обмениваться информацией, на каждом из них должен быть создан объект сокета.

Компьютер, принимающий соединение, (сервер) должен присвоить своему объекту сокета определенный номер порта. *Порт* – это 16-битное число в диапазоне 0 – 65 535, которое используется клиентами для уникальной идентификации серверов. Порты с номерами 0 – 1023 зарезервированы для нужд системы и используются наиболее распространенными сетевыми протоколами. Ниже перечислены некоторые распространенные протоколы с присвоенными им номерами портов (более полный список можно найти по адресу: *http://www.iana.org/assignments/port-numbers*):

Служба	Номер порта
FTP-Data	20
FTP-Control	21
SSH	22
Telnet	23
SMTP (электронная почта)	25
HTTP (WWW)	80
IMAP	143
HTTPS (безопасная WWW)	443

Процедура установки **TCP-соединения между клиентом и сервером определяется** точной последовательностью операций, как показано на рис. 21.1.

На стороне сервера, работающего по протоколу **TCP, объект сокета, используемый** для приема запросов на соединение, – это не тот же самый сокет, что в дальнейшем используется для обмена данными с клиентом. В частности, системный вызов `accept()` возвращает новый объект сокета, который фактически будет использоваться для обслуживания соединения. Это позволяет серверу одновременно обслуживать соединения с большим количеством клиентов.

Взаимодействия по протоколу UDP выполняются похожим способом, за исключением того, что клиенты и серверы не устанавливают логическое соединение друг с другом, как показано на рис. 21.2.

Следующий пример иллюстрирует применение протокола TCP клиентом и сервером, использующими модуль `socket`. В этом примере сервер просто возвращает клиенту текущее время в виде строки.

```
# Программа сервера времени
from socket import *
import time
```

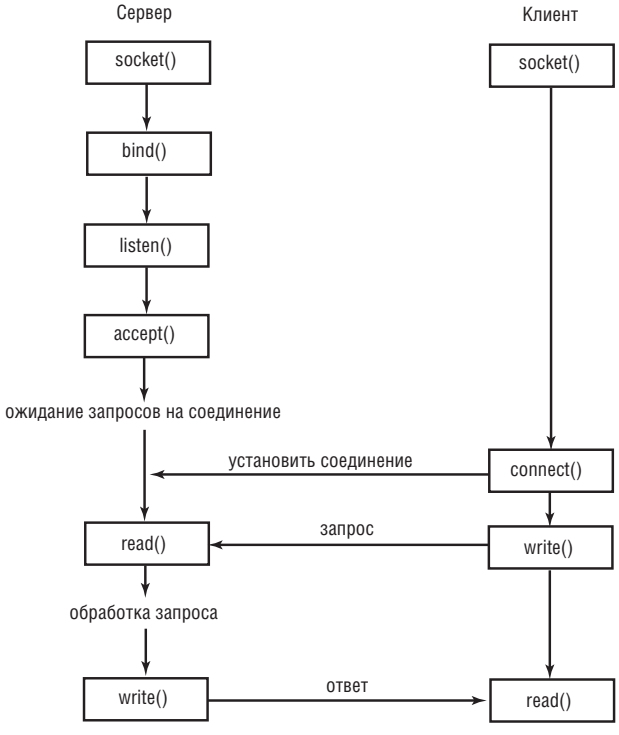


Рис. 21.1. *Процедура установки TCP-соединения*

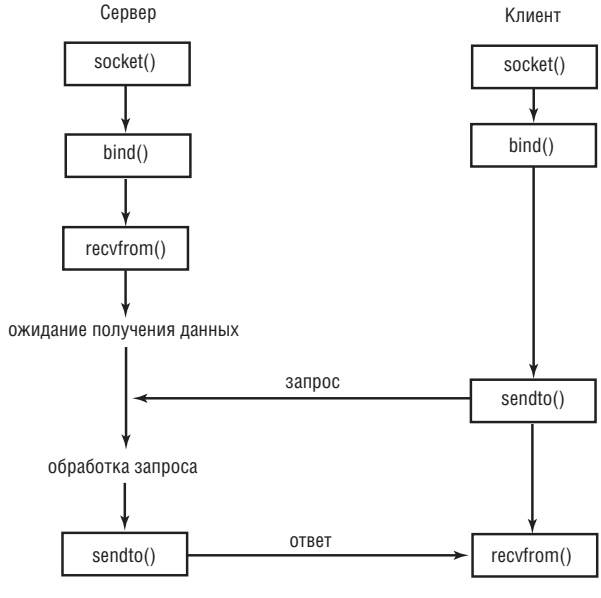


Рис. 21.2. *Обмен данными по протоколу UDP*

```
s = socket(AF_INET, SOCK_STREAM)  # Создает сокет TCP
s.bind(('',8888))                 # Присваивает порт 8888
s.listen(5)                       # Переходит в режим ожидания запросов;
                                  # одновременно обслуживает не более
                                  # 5 запросов.
while True:
    client,addr = s.accept()      # Принять запрос на соединение
    print("Получен запрос на соединение с %s" % str(addr))
    timestr = time.ctime(time.time()) + "\r\n"
    client.send(timestr.encode('ascii'))
    client.close()
```

Ниже приводится клиентская программа:

```
# Программа клиента, запрашивающего текущее время
from socket import *
s = socket(AF_INET,SOCK_STREAM)  # Создать сокет TCP
s.connect(('localhost', 8888))   # Соединиться с сервером
tm = s.recv(1024)                # Принять не более 1024 байтов данных
s.close()
print("Текущее время: %s" % tm.decode('ascii'))
```

Пример реализации обмена данными по протоколу UDP приводится в разделе с описанием модуля `socket` ниже, в этой главе.

В сетевых протоколах обмен данными часто выполняется в текстовой форме. Поэтому особое внимание необходимо уделять кодировке текста. В Python 3 все строки состоят из символов Юникода, что влечет за собой необходимость кодировать строки, передаваемые через сеть. Именно по этой причине в программе сервера к отправляемым данным применяется метод `encode('ascii')`. Точно так же, когда клиент принимает данные из сети, эти данные поступают в виде простой последовательности кодированных байтов. Если вывести эту последовательность на экран или попробовать интерпретировать ее как текст, результат, скорее всего, получится совсем не тот, какого вы ожидали. Поэтому прежде чем работать с данными, их необходимо декодировать. Для этого в программе клиента применяется метод `decode('ascii')` к принимаемым данным.

В оставшейся части главы описываются модули, имеющие отношение к программированию с применением сокетов. В главе 22 описываются высокоуровневые модули, обеспечивающие поддержку различных интернет-приложений, таких как электронная почта и Веб.

Модуль asynchat

Модуль `asynchat` упрощает реализацию приложений, в которых используются асинхронные сетевые операции, поддерживаемые модулем `asyncore`. Это достигается за счет обертывания низкоуровневых операций ввода-вывода, реализованных в модуле `asyncore`, высокоуровневым программным интерфейсом, предназначенным для работы с сетевыми протоколами, основанными на простых механизмах типа «запрос-ответ» (например, HTTP).

Для работы с этим модулем необходимо определить класс, производный от класса async_chat. Внутри этого класса необходимо определить два метода: collect_incoming_data() и found_terminator(). Первый метод вызывается всякий раз, когда через сетевое соединение поступают какие-либо данные. Обычно этот метод просто принимает данные и сохраняет их. Метод found_terminator() вызывается, когда будет обнаружен конец запроса. Например, при использовании протокола HTTP признаком конца запроса является пустая строка.

Для вывода данных объекты класса async_chat поддерживают выходную очередь FIFO. Если программе потребуется отправить данные, ей достаточно просто добавить их в очередь. Когда операция записи в сетевое соединение станет возможной, данные из этой очереди будут отправлены автоматически.

`async_chat([`*sock*`])`

Базовый класс. Используется для создания новых обработчиков. Класс async_chat является производным от класса asyncore.dispatcher и предоставляет те же методы. В аргументе *sock* передается объект сокета, который будет использоваться для обмена данными.

Экземпляр *a* класса async_chat обладает следующими методами помимо тех, что уже предоставляются базовым классом asyncore.dispatcher:

a`.close_when_done()`

Сообщает об окончании последовательности исходящих данных, добавляя None в выходную очередь FIFO. Когда процедура записи обнаружит это значение, она закроет канал.

a`.collect_incoming_data(`*data*`)`

Вызывается всякий раз при поступлении очередной порции данных. В аргументе *data* передаются полученные данные, которые обычно сохраняются для последующего использования. Этот метод должен быть реализован пользователем.

a`.discard_buffers()`

Уничтожает все данные, хранящиеся в буферах ввода-вывода и в выходной очереди FIFO.

a`.found_terminator()`

Вызывается, когда обнаруживается признак конца данных, установленный методом set_terminator(). Этот метод должен быть реализован пользователем. Обычно в этом методе выполняется обработка данных, собранных методом collect_incoming_data().

a`.get_terminator()`

Возвращает признак окончания последовательности данных.

a`.push(`*data*`)`

Добавляет данные в выходную очередь FIFO. В аргументе *data* передается строка с исходящими данными.

a.push_with_producer(*producer*)

Добавляет объект поставщика *producer* в выходную очередь FIFO. Объект *producer* может быть любым объектом, имеющим метод more(). Метод more() должен возвращать строку при каждом вызове. Пустая строка служит признаком окончания данных. За кулисами объект класса async_chat в цикле будет вызывать метод more(), чтобы получить данные для записи в канал вывода. В очередь FIFO можно добавить несколько объектов поставщиков, многократно вызвав метод push_with_producer().

s.set_terminator(*term*)

Устанавливает признак окончания данных. Аргумент *term* может быть строкой, целым числом или объектом None. Если в аргументе *term* передается строка, всякий раз, когда она будет встречаться во входных данных, будет вызываться метод found_terminator(). Если в аргументе *term* передается целое число, оно интерпретируется, как счетчик байтов. Всякий раз, когда будет принято указанное количество байтов, будет вызываться метод found_terminator(). Если в аргументе *term* передается None, прием данных будет осуществляться до бесконечности.

В модуле имеется класс, который может передаваться методу *a*.push_with_producer() для воспроизводства данных.

simple_producer(*data* [, *buffer_size*])

Создает простой объект поставщика, который делит строку байтов *data* на фрагменты. Аргумент *buffer_size* определяет размер фрагмента и по умолчанию принимает значение 512.

Модуль asynchat всегда используется совместно с модулем asyncore. Например, модуль asyncore может использоваться для создания сервера, принимающего входящие соединения, а модуль asynchat — для реализации обработчиков соединений. В следующем примере демонстрируется, как реализовать минимально возможный веб-сервер, обрабатывающий запросы GET. В примере почти полностью отсутствуют проверки на наличие ошибок и другие особенности, но его должно быть достаточно для начала. Сравните этот пример с примером, который приводится ниже, в разделе с описанием модуля asyncore.

```
# Асинхронный сервер HTTP, реализованный на основе модуля asynchat
import asynchat, asyncore, socket
import os
import mimetypes
try:
    from http.client import responses     # Python 3
except ImportError:
    from httplib import responses         # Python 2

# Следующий класс включает модуль asyncore
# и просто принимает входящие соединения
class async_http(asyncore.dispatcher):
    def __init__(self,port):
        asyncore.dispatcher.__init__(self)
        self.create_socket(socket.AF_INET,socket.SOCK_STREAM)
```

```
        self.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
        self.bind((‘’,port))
        self.listen(5)

    def handle_accept(self):
        client,addr = self.accept()
        return async_http_handler(client)

# Следующий класс обслуживает асинхронные запросы HTTP.
class async_http_handler(asynchat.async_chat):
    def __init__(self,conn=None):
        asynchat.async_chat.__init__(self,conn)
        self.data = []
        self.got_header = False
        self.set_terminator(b"\r\n\r\n")

    # Принимает входящие данные и добавляет их в буфер
    def collect_incoming_data(self,data):
        if not self.got_header:
            self.data.append(data)

    # Обрабатывает признак конца данных (пустая строка)
    def found_terminator(self):
        self.got_header  = True
        header_data      = b"".join(self.data)
        # Преобразовать данные заголовка (двоичные) в текст
        # для последующей обработки
        header_text      = header_data.decode(‘latin-1’)
        header_lines     = header_text.splitlines()
        request          = header_lines[0].split()
        op               = request[0]
        url              = request[1][1:]
        self.process_request(op,url)

    # Добавляет текст в исходящий поток, предварительно кодируя его
    def push_text(self,text):
        self.push(text.encode(‘latin-1’))

    # Обрабатывает запрос
    def process_request(self, op, url):
        if op == “GET”:
            if not os.path.exists(url):
                self.send_error(404,"File %s not found\r\n")
            else:
                type, encoding = mimetypes.guess_type(url)
                size = os.path.getsize(url)
                self.push_text(“HTTP/1.0 200 OK\r\n")
                self.push_text(“Content-length: %s\r\n” % size)
                self.push_text(“Content-type: %s\r\n” % type)
                self.push_text(“\r\n")
                self.push_with_producer(file_producer(url))
        else:
            self.send_error(501,"%s method not implemented” % op)
        self.close_when_done()
```

```
        # Обработка ошибок
        def send_error(self,code,message):
            self.push_text("HTTP/1.0 %s %s\r\n" % (code, responses[code]))
            self.push_text("Content-type: text/plain\r\n")
            self.push_text("\r\n")
            self.push_text(message)

    class file_producer(object):
        def __init__(self,filename,buffer_size=512):
            self.f = open(filename,"rb")
            self.buffer_size = buffer_size
        def more(self):
            data = self.f.read(self.buffer_size)
            if not data:
                self.f.close()
            return data

    a = async_http(8080)
    asyncore.loop()
```

Чтобы опробовать этот пример, необходимо указывать адреса URL, соответствующие файлам, находящимся в том же каталоге, где будет запущен сервер.

Модуль asyncore

Модуль `asyncore` используется для разработки сетевых приложений, в которых события, связанные с сетью, обрабатываются асинхронно, как последовательность событий, рассылаемых циклом событий, построенным на основе системного вызова `select()`. Такой подход удобно использовать в сетевых программах, реализующих многозадачность без использования потоков управления или процессов. Этот прием способен обеспечить высокую производительность при коротких операциях. Все функциональные возможности этого модуля представлены классом `dispatcher`, который является тонкой оберткой вокруг обычного объекта сокета.

`dispatcher([`*sock*`])`

Базовый класс, определяющий неблокирующий объект сокета, управляемый событиями. В аргументе *sock* передается существующий объект сокета. Если конструктор вызывается без аргумента, позднее необходимо будет создать сокет вызовом метода `create_socket()` (описывается ниже). После его создания сетевые события будут обрабатываться специальными методами-обработчиками. Кроме того, все созданные объекты класса `dispatcher` сохраняются во внутреннем списке, который используется некоторыми функциями опроса.

Для обработки сетевых событий вызываются следующие методы объектов класса `dispatcher`. Они должны быть определены в классах, производных от класса `dispatcher`.

d`.handle_accept()`

Вызывается объектом сокета, принимающим соединения, когда поступает новый запрос на соединение.

d.handle_close()

Вызывается при закрытии сокета.

d.handle_connect()

Вызывается после того, как соединение будет установлено.

d.handle_error()

Вызывается, когда появляется необработанное исключение.

d.handle_expt()

Вызывается, когда сокет получает срочные данные.

d.handle_read()

Вызывается, когда появляются новые данные, доступные для чтения из сокета.

d.handle_write()

Вызывается, когда выполняется операция записи данных.

d.readable()

Этот метод используется циклом select(), чтобы посмотреть, готов ли объект записывать данные. Возвращает True, если готов, и False – в противном случае. Этот метод вызывается, чтобы определить, должен ли вызываться метод handle_read(), чтобы сообщить о получении новых данных.

d.writable()

Вызывается циклом select(), чтобы посмотреть, готов ли объект записывать данные. Возвращает True, если готов, и False – в противном случае. Этот метод всегда вызывается, чтобы определить, должен ли вызываться метод handle_write(), чтобы произвести операцию записи.

Вдобавок к предыдущим методам для выполнения низкоуровневых операций над сокетом могут использоваться следующие методы. Они похожи на методы объекта сокета.

d.accept()

Принимает соединение. Возвращает кортеж (*client*, *addr*), где в поле *client* возвращается объект сокета, используемый для обмена данными через соединение, а в поле *addr* – адрес клиента.

d.bind(*address*)

Присваивает сокету указанный адрес *address*. В аргументе *address* обычно передается кортеж (*host*, *port*), однако точное представление адреса зависит от используемого семейства адресов.

d.close()

Закрывает сокет.

d.connect(*address*)

Устанавливает соединение. В аргументе *address* передается кортеж (*host*, *port*).

d.create_socket(*family*, *type*)

Создает новый сокет. Аргументы имеют тот же смысл, что и в функции socket.socket().

d.listen([*backlog*])

Принимает входящие соединения. В аргументе *backlog* передается целое число, которое передается функции socket.listen(), на основе которой реализован этот метод.

d.recv(*size*)

Принимает до *size* байтов. Пустая строка служит признаком того, что клиент закрыл канал.

d.send(*data*)

Отправляет данные *data*. В аргументе *data* передается строка байтов.

Следующая функция используется для запуска цикла приема и обработки событий:

loop([*timeout* [, *use_poll* [, *map* [, *count*]]]])

Запускает бесконечный цикл опроса. Если в аргументе *use_poll* передается значение False, опрос выполняется с помощью функции select(), в противном случае используется функция poll(). Аргумент *timeout* определяет предельное время ожидания и по умолчанию принимает значение 30 секунд. В аргументе *map* передается словарь со всеми каналами для мониторинга. Аргумент *count* определяет количество операций опроса, которые должны быть выполнены перед тем, как функция вернет управление. Если в аргументе *count* передать None (по умолчанию), функция loop() выполняет бесконечный цикл опроса, пока все каналы не будут закрыты. Если в аргументе *count* передать 1, функция выполнит одну проверку на наличие событий и вернет управление.

Пример

Следующий пример реализует веб-сервер с минимальными возможностями на основе модуля asyncore. В примере определяются два класса: asynhttp, принимающий соединения, и asynclient, обрабатывающий запросы клиентов. Сравните этот пример с примером, который приводился в разделе с описанием модуля asynchat. Основное отличие этого примера состоит в том, что он реализован на более низком уровне, из-за чего потребовалось побеспокоиться о таких проблемах, как разбиение потока входных данных на строки, буферизация данных и идентификация пустых строк, завершающих заголовки запросов.

```
# Асинхронный сервер HTTP
import asyncore, socket
import os
import mimetypes
import collections
try:
    from http.client import responses     # Python 3
```

```
except ImportError:
    from httplib import responses         # Python 2

# Следующий класс просто принимает входящие соединения
class async_http(asyncore.dispatcher):
    def __init__(self,port):
        asyncore.dispatcher.__init__(self)
        self.create_socket(socket.AF_INET,socket.SOCK_STREAM)
        self.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
        self.bind((‘’,port))
        self.listen(5)

    def handle_accept(self):
        client,addr = self.accept()
        return async_http_handler(client)

# Класс, обслуживающий клиентов
class async_http_handler(asyncore.dispatcher):
    def __init__(self, sock = None):
        asyncore.dispatcher.__init__(self,sock)
        self.got_request    = False               # Запрос HTTP прочитан?
        self.request_data   = b””
        self.write_queue    = collections.deque()
        self.responding     = False

    # Может использоваться для чтения, только если запрос еще не был прочитан
    def readable(self):
        return not self.got_request

    # Читает входящие данные запроса
    def handle_read(self):
        chunk = self.recv(8192)
        self.request_data += chunk
        if b’\r\n\r\n’ in self.request_data:
            self.handle_request()

    # Обрабатывает входящий запрос
    def handle_request(self):
        self.got_request = True
        header_data  = self.request_data[:self.request_data.find(b’\r\n\r\n’)]
        header_text  = header_data.decode(‘latin-1’)
        header_lines = header_text.splitlines()
        request      = header_lines[0].split()
        op           = request[0]
        url          = request[1][1:]
        self.process_request(op,url)

    # Обрабатывает запрос
    def process_request(self,op,url):
        self.responding = True
        if op == “GET”:
            if not os.path.exists(url):
                self.send_error(404,”File %s not found\r\n” % url)
            else:
                type, encoding = mimetypes.guess_type(url)
```

```
                size = os.path.getsize(url)
                self.push_text('HTTP/1.0 200 OK\r\n')
                self.push_text('Content-length: %d\r\n' % size)
                self.push_text('Content-type: %s\r\n' % type)
                self.push_text('\r\n')
                self.push(open(url,"rb").read())
            else:
                self.send_error(501,"%s method not implemented" % self.op)

        # Обработка ошибок
        def send_error(self,code,message):
            self.push_text('HTTP/1.0 %s %s\r\n' % (code, responses[code]))
            self.push_text('Content-type: text/plain\r\n')
            self.push_text('\r\n')
            self.push_text(message)

        # Добавляет двоичные данные в выходную очередь
        def push(self,data):
            self.write_queue.append(data)

        # Добавляет текстовые данные в выходную очередь
        def push_text(self,text):
            self.push(text.encode('latin-1'))

        # Готов для записи, только если ответ готов
        def writable(self):
            return self.responding and self.write_queue

        # Записывает данные ответа
        def handle_write(self):
            chunk = self.write_queue.popleft()
            bytes_sent = self.send(chunk)
            if bytes_sent != len(chunk):
                self.write_queue.appendleft(chunk[bytes_sent:])
            if not self.write_queue:
                self.close()

    # Создать сервер
    a = async_http(8080)
    # Запустить бесконечный цикл опроса
    asyncore.loop()
```

См. также

Описание модулей socket (стр. 586), select (стр. 572), SocketServer (стр. 611), пакета http (стр. 623).

Модуль select

Модуль `select` предоставляет доступ к системным вызовам `select()` и `poll()`. Системный вызов `select()` обычно используется для реализации опроса, или мультиплексирования, обработки нескольких потоков ввода-вывода, без использования потоков управления или дочерних процессов. В системах UNIX эти вызовы можно использовать для работы с файлами, сокета-

ми, каналами и со многими другими типами файлов. В Windows их можно использовать только для работы с сокетами.

`select(`*`iwtd`*`, `*`owtd`*`, `*`ewtd`*` [, `*`timeout`*`])`

Запрашивает информацию о готовности к вводу, выводу и о наличии исключений для группы дескрипторов файлов. В первых трех аргументах передаются списки с целочисленными дескрипторами файлов или с объектами, обладающими методом `fileno()`, который возвращает дескриптор файла. Аргумент *`iwtd`* определяет список объектов, которые проверяются на готовность к вводу, *`owtd`* – список объектов, которые проверяются на готовность к выводу, и *`ewtd`* – список объектов, которые проверяются на наличие исключительных ситуаций. В любом из аргументов допускается передавать пустой список. В аргументе *`timeout`* передается число с плавающей точкой, определяющее предельное время ожидания в секундах. При вызове без аргумента *`timeout`* функция ожидает, пока хотя бы один из дескрипторов не окажется в требуемом состоянии. Если в этом аргументе передать число `0`, функция просто выполнит опрос и тут же вернет управление. Возвращает кортеж списков с объектами, находящимися в требуемом состоянии. Эти списки включают подмножества объектов в первых трех аргументах. Если к моменту истечения предельного времени ожидания ни один из дескрипторов не находится в требуемом состоянии, возвращается три пустых списка. В случае ошибки возбуждается исключение `select.error`. В качестве значения исключения возвращается та же информация, что и в исключениях `IOError` и `OSError`.

`poll()`

Создает объект, выполняющий опрос с помощью системного вызова `poll()`. Эта функция доступна только в системах, поддерживающих системный вызов `poll()`.

Объект *`p`*, возвращаемый функцией `poll()`, поддерживает следующие методы:

`p``.register(`*`fd`*` [, `*`eventmask`*`])`

Регистрирует новый дескриптор файла *`fd`*. В аргументе *`fd`* может передаваться целочисленный дескриптор или объект, обладающий методом `fileno()`, с помощью которого можно получить дескриптор. В аргументе *`eventmask`* передается битная маска, составленная с помощью битовой операции ИЛИ из следующих флагов, которая определяет интересующие события:

Константа	Описание
`POLLIN`	Имеются данные, доступные для чтения.
`POLLPRI`	Имеются срочные данные, доступные для чтения.
`POLLOUT`	Готов к записи.
`POLLERR`	Ошибка.
`POLLHUP`	Разрыв соединения.
`POLLNVAL`	Недопустимый запрос.

При вызове функции без аргумента *eventmask* проверяются события POLLIN, POLLPRI и POLLOUT.

p.unregister(*fd*)

Удаляет дескриптор файла *fd* из объекта, выполняющего опрос. Возбуждает исключение KeyError, если дескриптор не был зарегистрирован ранее.

p.poll([*timeout*])

Проверяет наступление событий для всех зарегистрированных дескрипторов. Необязательный аргумент *timeout* определяет предельное время ожидания в миллисекундах. Возвращает список кортежей (*fd, event*), где поле *fd* является дескриптором файла, а поле *event* – битной маской, определяющей события. Поля этой битовой маски соответствуют константам POLLIN, POLLOUT и так далее. Например, чтобы проверить наличие события POLLIN, достаточно просто проверить значение выражение *event* & POLLIN на неравенство нулю. Если возвращается пустой список, это означает, что в течение указанного времени ожидания не возникло ни одного события.

Дополнительные возможности модуля

Функции select() и poll(), объявленные в этом модуле, совместимы со многими операционными системами. Кроме того, в системах Linux модуль select предоставляет интерфейс к механизму определения состояния по перепаду и по значению (epoll), который может обеспечить значительно более высокую производительность. В системах BSD предоставляется возможность доступа к очереди ядра и к объектам событий. Описание этих программных интерфейсов можно найти в электронной документации к модулю select, по адресу *http://docs.python.org/library/select*.

Усложненный пример использования асинхронного ввода-вывода

Иногда модуль select используется для реализации серверов, основанных на тасклетах и сопрограммах – механизмах многозадачности, в которых не используются потоки управления или процессы. Следующий пример иллюстрирует данную концепцию, реализуя диспетчер задач для сопрограмм, основанный на операциях ввода-вывода. Предупреждаю, что это самый сложный пример в книге и вам может потребоваться внимательно исследовать его, чтобы понять, как он работает. Возможно, вам также потребуется ознакомиться с моим учебным руководством «A Curious Course on Coroutines and Concurrency» (*http://www.dabeaz.com/coroutines*), где можно найти дополнительный справочный материал.

```
import select
import types
import collections

# Объект, представляющий запущенную задачу
class Task(object):
    def __init__(self,target):
        self.target  = target # Сопрограмма
```

```
        self.sendval = None   # Значение, которое передается при возобновлении
        self.stack   = []     # Стек вызовов

    def run(self):
        try:
            result = self.target.send(self.sendval)
            if isinstance(result,SystemCall):
                return result
            if isinstance(result,types.GeneratorType):
                self.stack.append(self.target)
                self.sendval = None
                self.target = result
            else:
                if not self.stack: return
                self.sendval = result
                self.target = self.stack.pop()
        except StopIteration:
            if not self.stack: raise
            self.sendval = None
            self.target = self.stack.pop()

# Объект, представляющий "системный вызов"
class SystemCall(object):
    def handle(self,sched,task):
        pass

# Объект диспетчера задач
class Scheduler(object):
    def __init__(self):
        self.task_queue    = collections.deque()
        self.read_waiting  = {}
        self.write_waiting = {}
        self.numtasks      = 0

    # Создает новую задачу из сопрограммы
    def new(self,target):
        newtask = Task(target)
        self.schedule(newtask)
        self.numtasks += 1

    # Добавляет задачу в очередь задач
    def schedule(self,task):
        self.task_queue.append(task)

    # Приостанавливает задачу, пока дескриптор файла не станет
    # доступным для чтения
    def readwait(self,task,fd):
        self.read_waiting[fd] = task

    # Приостанавливает задачу, пока дескриптор файла не станет
    # доступным для записи
    def writewait(self,task,fd):
        self.write_waiting[fd] = task

    # Главный цикл диспетчера задач
    def mainloop(self,count=-1,timeout=None):
```

```
        while self.numtasks:
            # Проверить наличие событий ввода-вывода
            if self.read_waiting or self.write_waiting:
                wait = 0 if self.task_queue else timeout
                r,w,e = select.select(self.read_waiting, self.write_waiting,
                                      [], wait)
                for fileno in r:
                    self.schedule(self.read_waiting.pop(fileno))
                for fileno in w:
                    self.schedule(self.write_waiting.pop(fileno))

            # Запустить все задачи, имеющиеся в очереди,
            # которые готовы к запуску
            while self.task_queue:
                task = self.task_queue.popleft()
                try:
                    result = task.run()
                    if isinstance(result,SystemCall):
                        result.handle(self,task)
                    else:
                        self.schedule(task)
                except StopIteration:
                    self.numtasks -= 1

            # Если нет задач, готовых для запуска,
            # требуется решить - продолжать или выйти
            else:
                if count > 0: count -= 1
                if count == 0:
                    return

# Реализация различных системных вызовов
class ReadWait(SystemCall):
    def __init__(self,f):
        self.f = f
    def handle(self,sched,task):
        fileno = self.f.fileno()
        sched.readwait(task,fileno)

class WriteWait(SystemCall):
    def __init__(self,f):
        self.f = f
    def handle(self,sched,task):
        fileno = self.f.fileno()
        sched.writewait(task,fileno)

class NewTask(SystemCall):
    def __init__(self,target):
        self.target = target
    def handle(self,sched,task):
        sched.new(self.target)
        sched.schedule(task)
```

Программный код этого примера реализует «операционную систему» в миниатюре. Ниже коротко описывается, как он действует:

- Вся основная работа выполняется сопрограммами. Сопрограммы, как и генераторы, используют инструкцию `yield`, но в отличие от генераторов, в сопрограммах она используется не только чтобы возвращать значения, но и чтобы принимать значения, которые передаются с помощью метода `send(`*`value`*`)`.
- Класс `Task` представляет готовую к запуску задачу и является всего лишь тонкой оберткой вокруг сопрограммы. Объект *`task`* класса `Task` может выполнять единственную операцию — *`task`*`.run()`. Этот метод возобновляет выполнение задачи, которая продолжает работать, пока не встретит следующую инструкцию `yield`, после чего выполнение задачи приостанавливается. После запуска задачи атрибут *`task`*`.sendval` содержит значение, которое должно быть послано соответствующему выражению `yield` в задаче. Задачи продолжают выполняться, пока не будет встречена следующая инструкция `yield`. Значение, которое воспроизводится этой инструкцией, определяет, что будет происходить дальше внутри задачи:
  - Если возвращаемым значением является другая сопрограмма (`type.GeneratorType`), это означает, что задача временно передает управление другой сопрограмме. Атрибут `stack` объекта класса `Task` представляет стек вызовов сопрограмм, в котором сохраняется прежняя сопрограмма. При следующем запуске задачи управление будет передано новой сопрограмме.
  - Если возвращаемым значением является экземпляр класса `SystemCall`, это означает, что задаче требуется, чтобы диспетчер выполнил некоторую операцию от своего имени (например, запустил бы новую задачу, приостановил бы выполнение задачи, пока не появится возможность выполнить операцию ввода-вывода, и так далее). Назначение этого объекта описывается чуть ниже.
  - Если возвращается какое-либо другое значение, это может означать одно из двух: либо управление вернула сопрограмма, запущенная из задачи, либо управление вернула сама задача. Если стек вызовов не пуст, это означает, что произошел возврат из сопрограммы, запущенной из задачи, поэтому вызывающая задача выталкивается из стека вызовов, а возвращаемое значение сохраняется, чтобы его можно было передать вызывающей сопрограмме. Вызывающая сопрограмма получит это значение, когда будет запущена в следующий раз. Если стек вызовов пуст, возвращаемое значение просто уничтожается.
  - Исключение `StopIteration` означает, что сопрограмма завершила свою работу. Когда появляется это исключение, управление передается предыдущей сопрограмме, сохраненной в стеке вызовов (если таковая имеется), либо исключение будет передано диспетчеру и в этом случае оно будет интерпретироваться, как признак завершения работы сопрограммы.
- Объект класса `SystemCall` представляет системный вызов. Когда работающей задаче требуется сообщить диспетчеру, что он должен выполнить операцию от своего имени, она с помощью инструкции `yield` возвращает

экземпляр класса SystemCall. Этот объект называется «системным вызовом», потому что он имитирует способ обращения к системным службам в настоящих многозадачных операционных системах, таких как UNIX или Windows. В частности, когда программе требуется обратиться к службе операционной системы, она передает управление системному вызову и предоставляет некоторую дополнительную информацию, необходимую для выполнения операции. В этом смысле возврат объекта класса SystemCall напоминает вызов системной «ловушки».

- Объект класса Scheduler представляет коллекцию объектов класса Task, которыми он управляет. Вся основная работа диспетчера построена вокруг очереди задач (атрибут task_queue), в которой хранятся задачи, готовые к запуску. Над очередью задач выполняются четыре основных операции. Метод new() принимает новую задачу, заворачивает ее в объект класса Task и помещает созданный объект в очередь. Метод schedule() получает существующий объект класса Task и вставляет его обратно в очередь. Метод mainloop() запускает цикл диспетчера, который обрабатывает задачи одну за другой, пока в очереди не останется задач. Методы readwait() и writewait() помещают объект класса Task во временную область ожидания, где он остается до появления ожидаемого события ввода-вывода. В этом случае работа задачи приостанавливается, но она не уничтожается, а просто бездействует в ожидании.
- Метод mainloop() является основой диспетчера задач. В самом начале этот метод проверяет, имеются ли задачи, ожидающие событий ввода-вывода. Если такие задачи имеются, диспетчер подготавливает и вызывает функцию select(), чтобы проверить наличие событий ввода-вывода. Если имеются какие-либо события, представляющие интерес, соответствующие задачи возвращаются обратно в очередь задач, чтобы появилась возможность запустить их. Затем метод mainloop() выталкивает очередную задачу из очереди и вызывает ее метод run(). Если какая-либо задача завершает работу (возбуждает исключение StopIteration), она уничтожается. Если задача просто возвращает управление, она опять добавляется в очередь задач, чтобы обеспечить возможность ее запуска в следующем цикле. Так продолжается до тех пор, пока либо не опустеет очередь задач, либо все задачи не окажутся приостановленными в ожидании событий ввода-вывода. Метод mainloop() может принимать дополнительный аргумент count, позволяющий обеспечить завершение работы метода после указанного количества операций опроса. Это может пригодиться, когда диспетчер встраивается в другой цикл обработки событий.
- Самым сложным аспектом диспетчера является обработка экземпляров класса SystemCall в методе mainloop(). Когда задача возвращает экземпляр класса SystemCall, диспетчер вызывает его метод handle(), передавая в качестве аргументов соответствующие экземпляры классов Scheduler и Task. Назначение системного вызова состоит в том, чтобы выполнить некоторую внутреннюю операцию, необходимую задаче или диспетчеру. Классы ReadWait(), WriteWait() и NewTask() являются примерами системных вызовов, которые приостанавливают выполнение задачи на время операции ввода-вывода или создают новую задачу. Например,

конструктор `ReadWait()` принимает задачу и вызывает метод `readwait()` диспетчера. После этого диспетчер принимает задачу и помещает ее в соответствующую область ожидания. Здесь можно видеть соблюдение очень важного принципа разделения объектов. Задачи возвращают объекты класса `SystemCall` службе обработки запросов, но они не вступают в прямое взаимодействие с диспетчером. Объекты класса `SystemCall`, в свою очередь, могут выполнять операции над задачами или диспетчерами, но они никак не привязаны к конкретной реализации диспетчера или задачи. Благодаря этому теоретически имеется возможность создавать совершенно разные реализации диспетчера (возможно даже с использование потоков управления), которые могли бы просто вставляться в существующий фреймворк и он при этом продолжал бы работать.

Ниже приводится пример простого сетевого сервера времени, реализованного с помощью данного диспетчера задач. Он поможет понять многое из того, о чем говорилось в предыдущем списке:

```
from socket import socket, AF_INET, SOCK_STREAM
def time_server(address):
    import time
    s = socket(AF_INET,SOCK_STREAM)
    s.bind(address)
    s.listen(5)
    while True:
        yield ReadWait(s)
        conn,addr = s.accept()
        print("Получен запрос на соединение с %s" % str(addr))
        yield WriteWait(conn)
        resp = time.ctime() + "\r\n"
        conn.send(resp.encode('latin-1'))
        conn.close()

sched = Scheduler()
sched.new(time_server(('',10000)))     # Сервер на порту 10000
sched.new(time_server(('',11000)))     # Сервер на порту 11000
sched.run()
```

В этом примере параллельно запускаются два сервера – каждый из них использует свой номер порта для приема соединений (это легко проверить с помощью утилиты `telnet`). Инструкции `yield ReadWait()` и `yield WriteWait()` вызывают приостановку сопрограммы каждого из серверов до момента, пока не появится возможность выполнить операцию ввода-вывода над соответствующим сокетом. Когда эти инструкции возвращают управление сопрограмме, немедленно выполняется соответствующая операция ввода-вывода, такая как `accept()` или `send()`.

Конструкции `ReadWait` и `WriteWait` могут показаться слишком низкоуровневыми. К счастью, архитектура приложения позволяет скрыть эти операции за кулисами библиотечных функций и методов, при условии, что они также будут являться сопрограммами. Взгляните на следующий объект, который служит оберткой вокруг объекта и имитирует его интерфейс:

```
class CoSocket(object):
    def __init__(self,sock):
```

```
        self.sock = sock
    def close(self):
        yield self.sock.close()
    def bind(self,addr):
        yield self.sock.bind(addr)
    def listen(self,backlog):
        yield self.sock.listen(backlog)
    def connect(self,addr):
        yield WriteWait(self.sock)
        yield self.sock.connect(addr)
    def accept(self):
        yield ReadWait(self.sock)
        conn, addr = self.sock.accept()
        yield CoSocket(conn), addr
    def send(self,bytes):
        while bytes:
            evt = yield WriteWait(self.sock)
            nsent = self.sock.send(bytes)
            bytes = bytes[nsent:]
    def recv(self,maxsize):
        yield ReadWait(self.sock)
        yield self.sock.recv(maxsize)
```

Ниже приводится реализация сервера времени, основанная на применении класса `CoSocket`:

```
from socket import socket, AF_INET, SOCK_STREAM
def time_server(address):
    import time
    s = CoSocket(socket(AF_INET,SOCK_STREAM))
    yield s.bind(address)
    yield s.listen(5)
    while True:
        conn,addr = yield s.accept()
        print(conn)
        print("Получен запрос на соединение с %s" % str(addr))
        resp = time.ctime()+"\r\n"
        yield conn.send(resp.encode('latin-1'))
        yield conn.close()

sched = Scheduler()
sched.new(time_server(('',10000)))   # Сервер на порту 10000
sched.new(time_server(('',11000)))   # Сервер на порту 11000
sched.run()
```

В этом примере программный интерфейс объекта класса `CoSocket` выглядит, как интерфейс обычного сокета. Единственное отличие состоит в том, что каждая операция должна предваряться инструкцией `yield` (так как все методы определены, как сопрограммы). На первый взгляд все это выглядит так странно, что заставляет задаться вопросом: а есть ли в этом какой-то смысл? Если запустить сервер, который приводится выше, можно заметить, что одновременно может выполняться несколько его копий без применения потоков управления или дочерних процессов. При этом про-

грамма выглядит, как «обычный» поток управления, если игнорировать все инструкции `yield`.

Ниже приводится пример асинхронного веб-сервера, который одновременно обслуживает несколько соединений с клиентами, но не использует при этом функции обратного вызова, потоки управления или процессы. Сравните его с примерами, реализованными на основе применения модулей `asynchat` и `asyncore`.

```
import os
import mimetypes
try:
    from http.client import responses   # Python 3
except ImportError:
    from httplib import responses       # Python 2
from socket import *

def http_server(address):
    s = CoSocket(socket(AF_INET,SOCK_STREAM))
    yield s.bind(address)
    yield s.listen(50)

    while True:
        conn,addr = yield s.accept()
        yield NewTask(http_request(conn,addr))
        del conn, addr

def http_request(conn,addr):
    request = b""
    while True:
        data = yield conn.recv(8192)
        request += data
        if b'\r\n\r\n' in request: break

    header_data    = request[:request.find(b'\r\n\r\n')]
    header_text    = header_data.decode('latin-1')
    header_lines   = header_text.splitlines()
    method, url, proto = header_lines[0].split()
    if method == 'GET':
        if os.path.exists(url[1:]):
            yield serve_file(conn,url[1:])
        else:
            yield error_response(conn,404,"File %s not found" % url)
    else:
        yield error_response(conn,501,"%s method not implemented" % method)
    yield conn.close()

def serve_file(conn,filename):
    content,encoding = mimetypes.guess_type(filename)
    yield conn.send(b"HTTP/1.0 200 OK\r\n")
    yield conn.send(("Content-type: %s\r\n" % content).encode('latin-1'))
    yield conn.send(("Content-length: %d\r\n" %
                     os.path.getsize(filename)).encode('latin-1'))
    yield conn.send(b"\r\n")
    f = open(filename,"rb")
```

```
        while True:
            data = f.read(8192)
            if not data: break
            yield conn.send(data)

    def error_response(conn,code,message):
        yield conn.send(("HTTP/1.0 %d %s\r\n" %
                         (code, responses[code])).encode('latin-1'))
        yield conn.send(b"Content-type: text/plain\r\n")
        yield conn.send(b"\r\n")
        yield conn.send(message.encode('latin-1'))

    sched = Scheduler()
    sched.new(http_server(('',8080)))
    sched.mainloop()
```

Внимательное изучение этого примера позволит надежно усвоить особенности приемов разработки многозадачных программ с применением сопрограмм, которые используются некоторыми весьма сложными сторонними модулями. Однако чрезмерное употребление этих приемов может привести к тому, что вас уволят с работы после очередной инспекции вашего программного кода.

Когда имеет смысл использовать асинхронные операции при работе с сетью

Использование асинхронных операций ввода-вывода (модули `asyncore` и `asynchat`), операций опроса и сопрограмм, как это демонстрировалось в предыдущих примерах, остается одним из самых таинственных аспектов программирования на языке Python. И все же эти приемы используются намного чаще, чем можно было бы подумать. Одной из часто упоминаемых причин использования асинхронных операций ввода-вывода является минимизация нагрузки, связанной с большим количеством потоков управления, особенно когда возникает необходимость управлять множеством клиентов при наличии ограничений, связанных с глобальной блокировкой интерпретатора (см. главу 20 «Потоки и многозадачность»).

Исторически модуль `asyncore` был одним из первых модулей в стандартной библиотеке, обеспечившим поддержку асинхронного ввода-вывода. Модуль `asynchat` появился немного позже с целью упростить использование модуля `asyncore`. Оба эти модуля используют подход, основанный на обработке событий ввода-вывода. Например, когда возникает событие ввода-вывода, вызывается функция обратного вызова. Функция обратного вызова реагирует на событие ввода-вывода и выполняет некоторую обработку данных. Если вам придется создавать крупные приложения в таком стиле, вы обнаружите, что обработка событий пронизывает практически все части приложения (например, события ввода-вывода приводят к вызову функций-обработчиков, которые в свою очередь вызывают другие функции-обработчики, и так до бесконечности). Этот подход используется в пакете Twisted (*http://twistedmatrix.com*), одном из наиболее популярных пакетов, предназначенных для разработки сетевых приложений.

Более современные решения опираются на применение сопрограмм, но они сложнее и используются реже, так как сопрограммы впервые появились в версии Python 2.5. Одна из важнейших особенностей сопрограмм состоит в том, что они позволяют писать приложения, которые больше походят на многопоточные программы. Например, в примерах реализации вебсервера не используются функции обратного вызова, и они выглядят очень похожими на программы, основанные на использовании потоков управления, – нужно просто привыкнуть к использованию инструкции `yield`. В интерпретаторе Python, не использующем стек вызовов (*http://www.stackless.com*), эта идея развита еще дальше.

Вообще говоря, в большинстве сетевых приложений не следует стремиться использовать приемы асинхронного ввода-вывода. Например, если потребуется создать сервер, который постоянно отправляет данные через сотни или даже тысячи соединений одновременно, применение методики, основанной на создании множества потоков управления, может обеспечить более высокую производительность. Это обусловлено тем, что производительность функции `select()` снижается тем больше, чем больше число соединений, которые нужно отслеживать. В операционной системе Linux этот недостаток можно устранить за счет использования специальных функций, таких как `epoll()`, но это ограничивает переносимость программного кода. Пожалуй, самую большую выгоду от использования асинхронного ввода-вывода можно получить в приложениях, где сетевые операции необходимо интегрировать в другие циклы событий (например, в цикл событий графического интерфейса) или где сетевые операции приходится добавлять в программный код, который выполняет массивные вычисления. В подобных ситуациях использование асинхронного ввода-вывода способно уменьшить время отклика.

Исключительно в целях демонстрации ниже приводится программа, выполняющая задачу, о которой поется в песне «10 миллионов бутылок пива на стене»:

```
bottles = 10000000

def drink_beer():
    remaining = 12.0
    while remaining > 0.0:
        remaining -= 0.1

def drink_bottles():
    global bottles
    while bottles > 0:
        drink_beer()
        bottles -= 1
```

Теперь предположим, что необходимо добавить возможность удаленного мониторинга, которая позволяла бы клиентам подключаться и следить за тем, сколько бутылок осталось. Один из вариантов состоит в том, чтобы запустить сервер в отдельном потоке управления, заставив его выполняться параллельно с основным приложением, как показано ниже:

```
def server(port):
    s = socket.socket(socket.AF_INET,socket.SOCK_STREAM)
    s.bind(('',port))
    s.listen(5)
    while True:
        client,addr = s.accept()
        client.send(("%d bottles\r\n" % bottles).encode('latin-1')
        client.close()

# Запуск сервера мониторинга
thr = threading.Thread(target=server,args=(10000,))
thr.daemon=True
thr.start()
drink_bottles()
```

Другой вариант состоит в том, чтобы реализовать сервер на основе опроса каналов ввода-вывода и внедрить операцию опроса непосредственно в главный цикл вычислений. Ниже приводится пример, в котором используется диспетчер сопрограмм, разработанный выше:

```
def drink_bottles():
    global bottles
    while bottles > 0:
        drink_beer()
        bottles -= 1
        scheduler.mainloop(count=1,timeout=0) # Проверить наличие соединений

# Асинхронный сервер, основанный на сопрограммах.
def server(port):
    s = CoSocket(socket.socket(socket.AF_INET,socket.SOCK_STREAM))
    yield s.bind(('',port))
    yield s.listen(5)
    while True:
        client,addr = yield s.accept()
        yield client.send(("%d bottles\r\n" % bottles).encode('latin-1')
        yield client.close()

scheduler = Scheduler()
scheduler.new(server(10000))
drink_bottles()
```

Если написать отдельную программу, которая периодически будет выполнять подключение к программе, имитирующей опустошение бутылок пива, и измерять время, необходимое на получение информации о количестве оставшихся бутылок, результаты могут оказаться весьма удивительными. На компьютере автора (макбук с двухъядерным процессором, работающим на частоте 2 ГГц) среднее время отклика сервера (измерения проводились по 1000 запросов), основанного на сопрограмме, составило примерно 1 миллисекунду, против 5 миллисекунд для сервера, основанного на потоках управления. Такая разница объясняется тем, что реализация, основанная на сопрограмме, способна отвечать сразу же, как только обнаруживает попытку установить соединение, тогда как многопоточный сервер не может быть запущен, пока не будет запланирован на выполнение

операционной системой. Учитывая наличие потока управления, выполняющего массивные вычисления, и глобальной блокировки интерпретатора, сервер вынужден простаивать, пока вычислительный поток управления не исчерпает выделенный ему квант времени. Во многих системах величина кванта времени составляет примерно 10 миллисекунд, то есть вышеупомянутое время отклика многопоточного сервера точно соответствует среднему арифметическому значению времени ожидания переключения вычислительного потока операционной системой.

Недостатком периодического опроса является чрезмерная нагрузка, если его выполнять слишком часто. Например, хотя время отклика в примере с опросом оказалось меньше, общее время выполнения программы увеличилось более чем на 50%. Если изменить реализацию так, что опрос будет проводиться только через каждые шесть бутылок пива, время отклика увеличится весьма незначительно и составит 1,2 миллисекунды, тогда как время выполнения программы будет всего на 3% больше, чем время выполнения программы без опроса. К сожалению, часто не бывает иного способа четко определить, как часто следует выполнять опрос, кроме как выполнив измерение производительности приложения.

Несмотря на то что уменьшение времени отклика кажется победой, реализация собственного механизма многозадачности может породить неприятные проблемы. Например, в задачах необходимо с особой осторожностью подходить к выполнению любых операций, которые могут вызывать приостановку процесса. В примере с веб-сервером имеется фрагмент программного кода, который открывает файл и читает из него данные. Эта операция может приостановить выполнение всей программы на достаточно продолжительный промежуток времени, если доступ к файлу будет сопряжен с необходимостью перемещения головок жесткого диска. Единственный способ устранить эту проблему состоит в том, чтобы дополнительно реализовать асинхронный доступ к файлу и добавить эту особенность в диспетчер задач. Для более сложных операций, таких как выполнение запросов к базе данных, определить, как реализовать асинхронный доступ, может оказаться намного сложнее. Один из возможных способов реализации таких операций – вынести их в отдельный поток управления и организовать обмен результатами с диспетчером задач по мере их поступления, что можно осуществить с помощью очередей сообщений. В некоторых системах существуют низкоуровневые системные вызовы, выполняющие асинхронные операции ввода-вывода (например, семейство функций `aio_*` в UNIX). К моменту написания этих строк в стандартной библиотеке языка Python еще не был реализован доступ к этим функциям, однако вы можете попробовать поискать сторонние модули, обеспечивающие такую возможность. По опыту автора, использование подобных функциональных возможностей намного сложнее, чем выглядит, а выигрыш от их добавления в программу на языке Python не стоит затрачиваемых усилий – часто в таких ситуациях более предпочтительно бывает использовать библиотеку для работы с потоками управления.

Модуль socket

Модуль socket предоставляет доступ к стандартному интерфейсу сокетов BSD. Хотя этот интерфейс разрабатывался для системы UNIX, тем не менее модуль socket может использоваться во всех платформах. Интерфейс сокетов разрабатывался для поддержки широкого разнообразия сетевых протоколов (IP, TIPC, Bluetooth и других). Однако наиболее часто используемым является протокол Интернета (Internet Protocol, IP), который включает в себя оба протокола, TCP и UDP. В языке Python поддерживаются обе версии протокола, IPv4 и IPv6, хотя версия IPv4 распространена намного шире.

Следует заметить, что этот модуль находится на достаточно низком уровне, обеспечивая непосредственный доступ к сетевым функциям, предоставляемым операционной системой. При разработке сетевых приложений может оказаться проще использовать модули, описываемые в главе 22, или модуль SocketServer, описываемый в конце этой главы.

Семейства адресов

Некоторые функции из модуля socket требуют указывать семейство адресов. Семейство определяет, какой сетевой протокол будет использоваться. Ниже приводится список констант, которые определены для этой цели:

Константа	Описание
AF_BLUETOOTH	Протокол Bluetooth
AF_INET	Протокол IPv4 (TCP, UDP)
AF_INET6	Протокол IPv6 (TCP, UDP)
AF_NETLINK	Протокол Netlink взаимодействия процессов
AF_PACKET	Пакеты канального уровня
AF_TIPC	Прозрачный протокол взаимодействия процессов (Transparent Inter-Process Communication protocol, TIPC)
AF_UNIX	Протоколы домена UNIX

Из них наиболее часто употребляются семейства AF_INET и AF_INET6, потому что они представляют стандартные сетевые соединения. Семейство AF_BLUETOOTH доступно только в системах, поддерживающих его (обычно это встраиваемые системы). Семейства AF_NETLINK, AF_PACKET и AF_TIPC поддерживаются только в операционной системе Linux. Семейство AF_NETLINK используется для обеспечения скоростных взаимодействий между пользовательскими процессами и ядром Linux. Семейство AF_PACKET используется для прямого взаимодействия с уровнем передачи данных (например, для непосредственной работы с пакетами ethernet). Семейство AF_TIPC определяет протокол, используемый для обеспечения скоростных взаимодействий процессов в кластерах, действующих под управлением операционной системы Linux (*http://tipc.sourceforge.net/*).

Типы сокетов

Некоторые функции в модуле socket дополнительно требуют указывать тип сокета. Тип сокета определяет тип взаимодействий (потоки или пакеты) для использования с указанным семейством протоколов. Ниже приводится список констант, используемых для этой цели:

Константа	Описание
SOCK_STREAM	Поток байтов с поддержкой логического соединения, обеспечивающего надежность передачи данных (TCP)
SOCK_DGRAM	Дейтаграммы (UDP)
SOCK_RAW	Простой сокет
SOCK_RDM	Дейтаграммы с надежной доставкой
SOCK_SEQPACKET	Обеспечивает последовательную передачу записей с поддержкой логических соединений

Наиболее часто в практике используются типы сокетов SOCK_STREAM и SOCK_DGRAM, потому что они соответствуют протоколам TCP и UDP в семействе протоколов Интернета (IP). Тип SOCK_RDM является разновидностью протокола UDP, обеспечивающей доставку дейтаграмм, но не гарантирующей порядок их доставки (дейтаграммы могут поступать получателю не в том порядке, в каком они отправлялись). Тип SOCK_SEQPACKET используется для отправки пакетов через соединения, ориентированные на передачу потока данных с сохранением порядка следования пакетов и их границ. Типы SOCK_RDM и SOCK_SEQPACKET не получили широкой поддержки, поэтому для обеспечения переносимости программ их лучше не использовать. Тип SOCK_RAW используется для низкоуровневого доступа к протоколу и может применяться для реализации специализированных операций, таких как отправка управляющих сообщений (например, сообщений ICMP). Обычно тип SOCK_RAW используется только в программах, которые предполагается выполнять с привилегиями суперпользователя или администратора.

Не все типы сокетов поддерживаются всеми семействами протоколов. Например, используя семейство AF_PACKET для перехвата пакетов ethernet в системе Linux, нельзя установить соединение для приема потока байтов, указав тип SOCK_STREAM. Вместо этого придется использовать тип SOCK_DGRAM или SOCK_RAW. Семейство AF_NETLINK поддерживает только тип SOCK_RAW.

Адресация

Чтобы обеспечить возможность взаимодействий через сокеты, необходимо указать адрес назначения. Форма адреса зависит от используемого семейства протоколов.

AF_INET (IPv4)

Для интернет-приложений, использующих протокол IPv4, адреса определяются в виде кортежа (*host*, *port*). Ниже приводятся два примера определения адресов:

```
('www.python.org', 80)
('66.113.130.182', 25)
```

Если в поле *host* передается пустая строка, это равносильно использованию константы INADDR_ANY, которая соответствует любому адресу. Обычно такой способ адресации используется на стороне сервера, когда создается сокет, принимающий соединения от любых клиентов. Если в поле *host* передается значение '<broadcast>', это равносильно использованию константы INADDR_BROADCAST в API сокетов.

Следует помнить, что при использовании имен хостов, таких как 'www.python.org', для их преобразования в IP-адреса будет использоваться служба доменных имен (DNS). В зависимости от настроек службы DNS программа может получать каждый раз различные IP-адреса. Чтобы избежать этого, в случае необходимости можно использовать обычные адреса, такие как '66.113.130.182'.

AF_INET6 (IPv6)

Для протокола IPv6 адреса указываются в виде кортежа из 4 элементов (*host*, *port*, *flowinfo*, *scopeid*). В адресах IPv6 поля *host* и *port* имеют тот же смысл, что и в адресах IPv4, за исключением того, что в числовой форме адрес IPv6 обычно задается строкой, состоящей из восьми шестнадцатеричных чисел, разделенных двоеточием, например: 'FEDC:BA98:7654:3210:FEDC:BA98:7654:3210' или '080A::4:1' (в последнем случае два двоеточия, следующие подряд, соответствуют недостающим группам чисел, состоящих из одних нулей).

В поле *flowinfo* указывается 32-битное число, содержащее 24-битный идентификатор потока (младшие 24 бита) и 4-битное значение приоритета (следующие 4 бита), старшие 4 бита зарезервированы. Идентификатор потока обычно используется, только когда отправителю необходимо разрешить специальные виды обработки трафика маршрутизаторами. В противном случае в поле *flowinfo* передается значение 0.

В поле *scopeid* указывается 32-битное число, которое требуется только при работе с адресами локальной сети и локального сайта. Адреса локальной сети всегда начинаются с префикса 'FE80:...' и используются для взаимодействий между компьютерами, находящимися в одной локальной сети (маршрутизаторы не передают локальные пакеты за пределы сети). В этом случае поле *scopeid* определяет индекс интерфейса, идентифицирующий конкретный сетевой интерфейс хоста. Эту информацию можно увидеть, воспользовавшись командой 'ifconfig' в UNIX или 'ipv6 if' – в Windows. Локальные адреса сайта всегда начинаются с префикса 'FEC0:...' и используются для взаимодействий между компьютерами, составляющими единый сайт (например, все компьютеры в подсети). В этом случае поле *scopeid* определяет числовой идентификатор сайта.

В случае отсутствия необходимости указывать значения полей *flowinfo* и *scopeid* адреса IPv6 можно указывать в виде кортежа (*host*, *port*), как и адреса IPv4.

AF_UNIX

При использовании сокетов домена UNIX адрес определяется как строка, содержащая путь в файловой системе, например: '/tmp/myserver'.

AF_PACKET

При использовании протокола пакетов в Linux адрес определяется, как кортеж (*device*, *protonum* [, *pkttype* [, *hatype* [, *addr*]]]), где в поле *device* передается строка, определяющая имя устройства, такая как "eth0", а в поле *protonum* – целое число, указывающее номер протокола ethernet, как определено в заголовочном файле <linux/if_ether.h> (например, число 0x0800 соответствует протоколу IP передачи пакетов). В поле *packet_type* передается целое число, определяющее тип пакетов, которое может принимать одно из следующих значений:

Константа	Описание
PACKET_HOST	Пакет, предназначенный для локального компьютера.
PACKET_BROADCAST	Широковещательный пакет физического уровня.
PACKET_MULTICAST	Пакет физического уровня для многоадресной передачи.
PACKET_OTHERHOST	Пакет, предназначенный для другого компьютера, который был перехвачен драйвером устройства, действующим в режиме прослушивания (promiscuous).
PACKET_OUTGOING	Пакет, исходящий из локального компьютера, который через петлевой интерфейс был передан пакетному сокету.

В поле *hatype* передается целое число, указывающее тип аппаратного адреса, используемого протоколом ARP, как определено в заголовочном файле <linux/if_arp.h>. В поле *addr* передается строка байтов с аппаратным адресом, структура которого определяется значением поля *hatype*. Для ethernet в поле *addr* передается строка с аппаратным адресом, состоящим из 6 байтов.

AF_NETLINK

При использовании протокола Netlink в Linux адрес определяется, как кортеж (*pid*, *groups*), где в обоих полях *pid* и *groups* передаются целые числа без знака. Поле *pid* – это индивидуальный адрес сокета; поле обычно содержит идентификатор процесса, создавшего сокет, или 0, если соединение устанавливается с ядром. Поле *groups* – это битная маска, определяющая группу связанных адресов. За дополнительной информацией обращайтесь к описанию протокола Netlink.

AF_BLUETOOTH

Вид адресов протоколов семейства Bluetooth определяется конкретным используемым протоколом. Для протокола L2CAP адрес определяется как кортеж (*addr*, *psm*), где в поле *addr* передается строка, такая как '01:23:45:67:89:ab', а в поле *psm* – целое число без знака. Для протокола RFCOMM адрес

определяется как кортеж (*addr*, *channel*), где в поле *addr* передается строка адреса, а в поле *channel* – целое число. Для протокола HCI адрес определяется как кортеж (*deviceno*,), где в поле *deviceno* передается целочисленный номер устройства. Для протокола SCO адрес определяется как строка *host*.

Константа BDADDR_ANY представляет любой адрес и имеет строковое значение '00:00:00:00:00:00'. Константа BDADDR_LOCAL имеет строковое значение '00:00:00:ff:ff:ff'.

AF_TIPC

Для семейства TIPC адрес определяется как кортеж (*addr_type*, *v1*, *v2*, *v3* [, *scope*]), где во всех полях передаются целые числа без знака. Поле *addr_type* может принимать одно из следующих значений, которые также могут передаваться в полях *v1*, *v2* и *v3*:

Тип адреса	Описание
TIPC_ADDR_NAMESEQ	*v1* – тип сервера, *v2* – идентификатор порта и *v3* – число 0.
TIPC_ADDR_NAME	*v1* – тип сервера, *v2* – младший номер порта и *v3* – старший номер порта.
TIPC_ADDR_ID	*v1* – узел, *v2* – ссылка и *v3* – число 0.

Необязательное поле *scope* может иметь одно из следующих значений: TIPC_ZONE_SCOPE, TIPC_CLUSTER_SCOPE или TIPC_NODE_SCOPE.

Функции

Ниже перечислены функции, которые определяются модулем socket:

`create_connection(`*`address`* `[, `*`timeout`*`])`

Устанавливает соединение типа SOCK_STREAM с адресом *address* и возвращает уже подключенный объект сокета. В аргументе *address* передается кортеж вида (*host*, *port*). Необязательный аргумент *timeout* определяет предельное время ожидания. Эта функция сначала вызывает функцию getaddrinfo() и затем пытается соединиться с каждым из возвращаемых ею адресов.

`fromfd(`*`fd`*`, `*`family`*`, `*`socktype`* `[, `*`proto`*`])`

На основе целочисленного дескриптора файла *fd* создает объект сокета. В остальных аргументах предаются семейство адресов, тип сокета и номер протокола, которые могут принимать те же значения, что и в функции socket(). Дескриптор файла должен ссылаться на ранее созданный объект сокета. Возвращает экземпляр класса SocketType.

`getaddrinfo(`*`host`*`, `*`port`* `[, `*`family`* `[, `*`socktype`* `[, `*`proto`* `[, `*`flags`*`]]]])`

Используя информацию *host* и *port* о хосте, эта функция возвращает список кортежей с информацией, необходимой для открытия соединения. В аргументе *host* передается строка с именем хоста или IP-адрес в числовой форме. В аргументе *port* передается число или строка, представляющая имя службы (например, "http", "ftp", "smtp"). Каждый из возвращае-

мых кортежей содержит пять элементов (*family*, *socktype*, *proto*, *canonname*, *sockaddr*). Поля *family*, *socktype* и *proto* могут принимать те же значения, которые передаются функции socket(). В поле *canonname* возвращается строка, представляющая каноническое имя хоста. В поле *sockaddr* возвращается кортеж с адресом сокета, как описывалось в предыдущем разделе, где описываются способы адресации. Например:

```
>>> getaddrinfo("www.python.org",80)
[(2,2,17,'',('194.109.137.226',80)), (2,1,6,'',('194.109.137.226'),80))]
```

В этом примере функция getaddrinfo() вернула информацию о двух возможных соединениях. Первый кортеж в списке соответствует соединению по протоколу UDP (*proto*=17), а второй – соединению по протоколу TCP (*proto*=6). Чтобы сузить набор доступных адресов, можно использовать дополнительный аргумент *flags* функции getaddrinfo(). Например, в следующем примере возвращается информация об установленном соединении по протоколу IPv4 TCP:

```
>>> getaddrinfo("www.python.org",80,AF_INET,SOCK_STREAM)
[(2,1,6,'',('194.109.137.226',80))]
```

Чтобы определить все доступные соединения для указанного семейства адресов, можно использовать специальную константу AF_UNSPEC. Например, в следующем примере выполняется запрос всех возможных адресов TCP, который может возвращать информацию как для версии IPv4, так и для версии IPv6:

```
>>> getaddrinfo("www.python.org","http", AF_UNSPEC, SOCK_STREAM)
[(2,1,6,'',('194.109.137.226',80))]
```

Функция getaddrinfo() предназначена для широкого применения и может применяться ко всем поддерживаемым сетевым протоколам (IPv4, IPv6 и так далее). Используйте эту функцию, если вас беспокоит проблема совместимости и поддержки протоколов, которые могут появиться в будущем, особенно если предполагается, что программа будет поддерживать протокол IPv6.

`getdefaulttimeout()`

Возвращает предельное время ожидания в секундах для сокетов. Значение None указывает, что предельное время ожидания не было определено.

`getfqdn([`*name*`])`

Возвращает полностью квалифицированную версию доменного имени *name*. Если функция вызывается без аргумента, она возвращает информацию о локальном компьютере. Например, вызов getfqdn("foo") мог бы вернуть строку "foo.quasievil.org".

`gethostbyname(`*hostname*`)`

Преобразует имя хоста, такое как 'www.python.org', в адрес IPv4. IP-адрес возвращается в виде строки, такой как '132.151.1.90'. Эта функция не поддерживает адреса IPv6.

`gethostbyname_ex(`*hostname*`)`

Преобразует имя хоста в адрес IPv4, но возвращает кортеж (*hostname*, *aliaslist*, *ipaddrlist*), где в поле *hostname* возвращается основное имя хоста, в поле *aliaslist* возвращается список альтернативных имен хоста для того же адреса, а в поле *ipaddrlist* – список адресов IPv4 для того же интерфейса на том же компьютере. Например, вызов gethostbyname_ex('www.python.org') может вернуть примерно такой результат: ('fang.python.org', ['www.python.org'], ['194.109.137.226']). Эта функция не поддерживает адреса IPv6.

`gethostname()`

Возвращает имя хоста локального компьютера.

`gethostbyaddr(`*`ip_address`*`)`

Возвращает ту же информацию, что и функция gethostbyname_ex(), по указанному IP-адресу, такому как '132.151.1.90'. Если в аргументе *ip_address* передать адрес IPv6, такой как 'FEDC:BA98:7654:3210:FEDC:BA98:7654:3210', функция вернет информацию, относящуюся к этому адресу IPv6.

`getnameinfo(`*`address`*`, `*`flags`*`)`

Преобразует адрес *address* сокета в кортеж (*host*, *port*), с учетом значения аргумента *flags*. В аргументе *address* передается кортеж, определяющий адрес, например: ('www.python.org',80). В аргументе *flags* передается битная маска, составленная из следующих флагов с помощью битовой операции ИЛИ:

Константа	Описание
NI_NOFQDN	Не использовать полностью квалифицированное имя для локального хоста.
NI_NUMERICHOST	Вернуть адрес в числовой форме.
NI_NAMEREQD	Требуется имя хоста. Возвращает признак ошибки, если для адреса *address* отсутствует запись DNS.
NI_NUMERICSERV	В поле *port* возвращается строка с номером порта.
NI_DGRAM	Указывает, что следует вернуть номер порта службы, поддерживающей протокол UDP, а не TCP (по умолчанию).

Основное назначение этой функции состоит в том, чтобы предоставить возможность получения дополнительной информации об адресе. Например:

```
>>> getnameinfo(('194.109.137.226',80),0)
('fang.python.org', 'http')
>>> getnameinfo(('194.109.137.226',80),NI_NUMERICSERV)
('fang.python.org','80')
```

`getprotobyname(`*`protocolname`*`)`

Преобразует имя протокола (такое как 'icmp') в номер протокола (такой как значение IPPROTO_ICMP), который можно будет передать функции socket() в третьем аргументе. Возбуждает исключение socket.error, если имя протокола не удается опознать. Обычно эта функция используется только для простых (raw) сокетов.

`getservbyname(`*`servicename`* `[,` *`protocolname`*`])`

Преобразует имя сетевой службы и имя протокола в номер порта этой службы. Например, вызов `getservbyname('ftp', 'tcp')` вернет число `21`. В необязательном аргументе *`protocolname`* допускается передавать либо значение `'tcp'`, либо `'udp'`. Возбуждает исключение `socket.error`, если имя *`servicename`* не соответствует ни одной известной службе.

`getservbyport(`*`port`* `[,` *`protocolname`*`])`

Выполняет операцию, обратную функции `getservbyname()`. По номеру порта в аргументе *`port`* эта функция возвращает строку с именем службы, если таковая определена. Например, вызов `getservbyport(21, 'tcp')` вернет строку `'ftp'`. В необязательном аргументе *`protocolname`* допускается передавать либо значение `'tcp'`, либо `'udp'`. Возбуждает исключение `socket.error`, если для указанного номера порта нет определенного имени службы.

`has_ipv6`

Логическая константа, которая принимает значение `True`, если система поддерживает протокол IPv6.

`htonl(`*`x`*`)`

Преобразует 32-битные целые числа с аппаратным порядком следования байтов в 32-битные целые числа с сетевым порядком следования байтов (прямой порядок, или big-endian).

`htons(`*`x`*`)`

Преобразует 16-битные целые числа с аппаратным порядком следования байтов в 16-битные целые числа с сетевым порядком следования байтов (прямой порядок, или big-endian).

`inet_aton(`*`ip_string`*`)`

Преобразует адреса IPv4, представленные в виде строки (например, `'135.128.11.209'`), в 32-битный упакованный двоичный формат для последующего использования в качестве 32-битового двоичного адреса. Возвращает строку из четырех двоичных байтов. Может использоваться для передачи адресов функциям, написанным на языке C, или когда требуется упаковать данные в структуру для последующей передачи другим программам. Не поддерживает адреса IPv6.

`inet_ntoa(`*`packedip`*`)`

Преобразует адрес IPv4 в двоичном упакованном формате в строку, с использованием привычной точечной нотации (например, `'135.128.11.209'`). В аргументе *`packedip`* передается строка из четырех байтов, содержащая 32-битный двоичный IP-адрес. Эта функция может использоваться, когда адрес возвращается функцией, написанной на языке C, или при распаковывании структур с данными. Не поддерживает адреса IPv6.

`inet_ntop(`*`address_family`*`,` *`packed_ip`*`)`

Преобразует IP-адрес *`packed_ip`* в двоичном упакованном формате в строку, такую как `'123.45.67.89'`. Аргумент *`address_family`* определяет семейство адресов и обычно в нем передается значение `AF_INET` или `AF_INET6`. Может

использоваться для получения строки с сетевым адресом из последовательности двоичных байтов (например, из содержимого низкоуровневого сетевого пакета).

`inet_pton(address_family, ip_string)`

Преобразует IP-адреса, такие как '123.45.67.89', в строку упакованных байтов. Аргумент *address_family* определяет семейство адресов, и обычно в нем передается значение `AF_INET` или `AF_INET6`. Эта функция может использоваться для вставки сетевых адресов в двоичном формате в двоичные пакеты данных.

`ntohl(x)`

Преобразует 32-битные целые числа с сетевым порядком следования байтов (прямой порядок, или big-endian) в 32-битные целые числа с аппаратным порядком следования байтов.

`ntohs(x)`

Преобразует 16-битные целые числа с сетевым порядком следования байтов (прямой порядок, или big-endian) в 16-битные целые числа с аппаратным порядком следования байтов.

`setdefaulttimeout(timeout)`

Устанавливает предельное время ожидания для вновь создаваемых объектов сокетов. В аргументе *timeout* передается число с плавающей точкой, определяющее интервал времени в секундах. Может передаваться значение `None`, чтобы указать на отсутствие предельного времени ожидания (по умолчанию).

`socket(family, type [, proto])`

По заданным значениям семейства адресов, типа сокета и номера протокола создает новый сокет. Аргумент *family* определяет семейство адресов, а аргумент *type* – тип сокета, как обсуждалось в первой части этого раздела. Для открытия соединения по протоколу TCP можно использовать вызов `socket(AF_INET, SOCK_STREAM)`. Для открытия соединения по протоколу UDP можно использовать вызов `socket(AF_INET, SOCK_DGRAM)`. Возвращает экземпляр класса `SocketType` (описывается ниже).

Аргумент с номером протокола, как правило, не передается (и по умолчанию принимается значение `0`). Обычно он используется только при создании простых сокетов (`SOCK_RAW`) и может принимать значение, зависящее от используемого семейства адресов. Ниже приводится список всех номеров протоколов, которые в языке Python могут использоваться совместно с семействами адресов `AF_INET` и `AF_INET6`, в зависимости от наличия их поддержки в системе:

Константа	Описание
`IPPROTO_AH`	Заголовок аутентификации в IPv6
`IPPROTO_BIP`	Протокол Banyan VINES (Banyan Virtual Network System, система распределенной сети)
`IPPROTO_DSTOPTS`	Параметры получателя IPv6

Константа	Описание
IPPROTO_EGP	Протокол внешней маршрутизации (EGP)
IPPROTO_EON	ISO CNLP (Connectionless Network Protocol, сетевой протокол без создания соединения)
IPPROTO_ESP	Протокол шифрования данных в IPv6
IPPROTO_ FRAGMENT	Заголовок фрагмента в IPv6
IPPROTO_GGP	Протокол межсетевого сопряжения (RFC 823)
IPPROTO_GRE	Общая инкапсуляция маршрутов (RFC 1701)
IPPROTO_HELLO	Протокол HELLO компании FuzzBall
IPPROTO_HOPOPTS	Параметры транзитных узлов в IPv6
IPPROTO_ICMP	IPv4 ICMP (Internet Control Message Protocol, протокол управляющих сообщений)
IPPROTO_ICMPV6	IPv6 ICMP
IPPROTO_IDP	XNS IDP (Xerox Network Services – Internet Datagram Protocol, сетевые службы Xerox – протокол дейтаграмм Интернета)
IPPROTO_IGMP	Протокол управления группами
IPPROTO_IP	IPv4
IPPROTO_IPCOMP	Протокол сжатия данных IP
IPPROTO_IPIP	Туннелирование IP через IP
IPPROTO_IPV4	Заголовок IPv4
IPPROTO_IPV6	Заголовок IPv6
IPPROTO_MOBILE	Мобильный протокол IP
IPPROTO_ND	Протокол Netdisk
IPPROTO_NONE	IPv6, заголовок «нет следующего пакета» (последний пакет в последовательности)
IPPROTO_PIM	Протокол независимой групповой передачи
IPPROTO_PUP	Универсальный протокол передачи пакетов компании Xerox (PARC Universal Packet, PUP)
IPPROTO_RAW	Низкоуровневый протокол IP
IPPROTO_ROUTING	Заголовок маршрутизации IPv6
IPPROTO_RSVP	Резервирование ресурса
IPPROTO_TCP	Протокол TCP
IPPROTO_TP	Транспортный протокол OSI (TP-4)
IPPROTO_UDP	Протокол UDP
IPPROTO_VRRP	Протокол резервирования виртуального маршрутизатора
IPPROTO_XTP	Протокол скоростной передачи

Ниже перечислены номера протоколов, используемых совместно с семейством адресов AF_BLUETOOTH:

Константа	Описание
BTPROTO_L2CAP	Протокол управления логическим каналом с самонастройкой
BTPROTO_HCI	Интерфейс хост-контроллера
BTPROTO_RFCOMM	Протокол замены кабеля
BTPROTO_SCO	Протокол синхронного канала с созданием логического соединения

`socketpair([`*family* `[, ` *type* `[, ` *proto* `]]])`

Создает пару объектов сокетов, использующих указанное семейство адресов *family*, тип *type* и протокол *proto*. Аргументы имеют тот же смысл, что и в функции socket(). Эта функция может использоваться только для создания сокетов домена UNIX (*family*=AF_UNIX). Аргумент *type* может принимать только значение SOCK_DGRAM или SOCK_STREAM. Если в аргументе *type* передается значение SOCK_STREAM, создается объект, известный как *канал потока*. В аргументе *proto* обычно передается значение 0 (по умолчанию). В основном эта функция используется для организации канала взаимодействия с процессами, которые создаются функцией os.fork(). Например, родительский процесс может с помощью функции socketpair() создать пару сокетов и вызвать функцию os.fork(). После этого родительский и дочерний процесс получают возможность взаимодействовать друг с другом, используя эти сокеты.

Сокеты в программе представлены экземплярами класса SocketType. Сокет *s* обладает следующими методами:

s`.accept()`

Принимает соединение и возвращает кортеж (*conn*, *address*), где в поле *conn* возвращается новый объект сокета, который может использоваться для приема и передачи данных через соединение, а в поле *address* возвращается адрес сокета с другой стороны соединения.

s`.bind(`*address*`)`

Присваивает сокету указанный адрес *address*. Формат представления адреса зависит от семейства адресов. В большинстве случаев это кортеж вида (*hostname*, *port*). Для IP-адресов пустая строка представляет INADDR_ANY (любой адрес), а строка '<broadcast>' представляет широковещательный адрес INADDR_BROADCAST. Значение INADDR_ANY (пустая строка) в поле *hostname* используется, чтобы показать, что сервер может принимать соединения на любом сетевом интерфейсе. Это значение часто используется, когда сервер имеет несколько сетевых интерфейсов. Значение INADDR_BROADCAST ('<broadcast>') в поле *hostname* применяется, когда сокет предполагается использовать для рассылки широковещательных сообщений.

s`.close()`

Закрывает сокет. Этот метод вызывается также, когда объект сокета утилизируется сборщиком мусора.

s.connect(*address*)

Устанавливает соединение с удаленным узлом, имеющим адрес *address*. Формат адреса *address* зависит от семейства, к которому он относится, но обычно в этом аргументе передается кортеж (*hostname*, *port*). В случае ошибки возбуждает исключение socket.error. При подключении к серверу, выполняющемуся на том же компьютере, в поле *hostname* можно передавать имя хоста 'localhost'.

s.connect_ex(*address*)

Устанавливает соединение, как и функция connect(*address*), но в случае успеха возвращает 0, а в случае ошибки – значение errno.

s.fileno()

Возвращает файловый дескриптор сокета.

s.getpeername()

Возвращает адрес удаленного узла, с которым установлено соединение. Обычно возвращает кортеж (*ipaddr*, *port*), но вообще формат возвращаемого значения зависит от используемого семейства адресов. Этот метод поддерживается не во всех системах.

s.getsockname()

Возвращает собственный адрес сокета. Обычно возвращаемым значением является кортеж (*ipaddr*, *port*).

s.getsockopt(*level*, *optname* [, *buflen*])

Возвращает значение параметра сокета. Аргумент *level* – **уровень параметра**. Для получения параметров уровня сокета в нем передается значение SOL_SOCKET, а для получения параметров уровня протокола – номер протокола, такой как IPPROTO_IP. Аргумент *optname* определяет имя параметра. Если аргумент *buflen* отсутствует, предполагается, что параметр имеет целочисленное значение, которое и возвращается методом. Если аргумент *buflen* указан, он определяет максимальный размер буфера, куда должно быть записано значение параметра. Этот буфер возвращается методом в виде строки байтов. Интерпретация содержимого буфера с помощью модуля struct или других инструментов целиком возлагается на вызывающую программу.

В следующей таблице перечислены параметры сокета, поддерживаемые в языке **Python. Большая часть этих параметров относится к расширенному** API сокетов и содержит низкоуровневую информацию о сети. Более подробное описание параметров можно найти в документации и в специализированной литературе. Имена типов, встречающиеся в столбце «Значение», соответствуют стандартным типам языка C, ассоциированным со значением и используемым в стандартном программном интерфейсе сокетов. Не все параметры поддерживаются в каждой из систем.

Ниже перечислены наиболее часто используемые имена параметров уровня SOL_SOCKET:

Имя параметра	Значение	Описание
SO_ACCEPTCONN	0, 1	Разрешает или запрещает прием соединения.
SO_BROADCAST	0, 1	Разрешает или запрещает передачу широковещательных сообщений.
SO_DEBUG	0, 1	Разрешает или запрещает запись отладочной информации.
SO_DONTROUTE	0, 1	Разрешает или запрещает передавать сообщения в обход таблицы маршрутизации.
SO_ERROR	int	Возвращает признак ошибки.
SO_EXCLUSIVEADDRUSE	0, 1	Разрешает или запрещает возможность присваивания того же адреса и порта другому сокету. Этот **параметр отключает действие параметра** SO_REUSEADDR.
SO_KEEPALIVE	0, 1	Разрешает или запрещает периодическую передачу служебных сообщений другой стороне, для поддержания соединения в активном состоянии.
SO_LINGER	linger	Откладывает вызов метода close(), **если в буфере передачи имеются данные.** Значение типа linger представляет собой упакованную двоичную строку, содержащую два 32-битных целых числа (*onoff*, *seconds*).
SO_OOBINLINE	0, 1	Разрешает или запрещает добавление внеочередных сообщений во входной поток.
SO_RCVBUF	int	Размер приемного буфера (в байтах).
SO_RCVLOWAT	int	Определяет минимальное количество байтов, которые должны быть прочитаны, прежде чем функция select() будет интерпретировать сокет, как доступный для чтения.
SO_RCVTIMEO	timeval	Максимальное время ожидания в секундах при приеме данных. Значение timeval представляет собой упакованную двоичную строку, содержащую два 32-битных целых числа без знака (*seconds*, *microseconds*).
SO_REUSEADDR	0, 1	Разрешает или запрещает повторное использование локальных адресов.
SO_REUSEPORT	0, 1	Разрешает или запрещает нескольким процессам присваивать сокетам один и тот же адрес, при условии, что этот параметр установлен в значение 1 во всех процессах.
SO_SNDBUF	int	Размер передающего буфера (в байтах).
SO_SNDLOWAT	int	Определяет, какое количество байтов должно остаться в передающем буфере, прежде чем функция select() будет интерпретировать сокет, как доступный для записи.

Имя параметра	Значение	Описание
SO_SNDTIMEO	timeval	Максимальное время ожидания в секундах при передаче данных. Описание значение типа timeval приводится в описании параметра SO_RCVTIMEO.
SO_TYPE	int	Тип сокета.
SO_USELOOPBACK	0, 1	Разрешает или запрещает маршрутизирующему сокету получать копии отправляемых им пакетов.

Следующие параметры доступны для уровня IPPROTO_IP:

Имя параметра	Значение	Описание
IP_ADD_MEMBERSHIP	ip_mreg	Включение в группу многоадресной передачи (доступен только для записи). Значение типа ip_mreg представляет собой упакованную двоичную строку, содержащую два 32-битных IP-адреса (*multiaddr*, localaddr), где число *multiaddr* определяет групповой адрес, а число *localaddr* – IP-адрес используемого локального интерфейса.
IP_DROP_MEMBERSHIP	ip_mreg	Выход из группы многоадресной передачи (доступен только для записи). Описание значения типа ip_mreg приводится выше.
IP_HDRINCL	int	Включается IP-заголовок.
IP_MAX_MEMBERSHIPS	int	Максимальное количество многоадресных групп.
IP_MULTICAST_IF	in_addr	Определяет исходящий интерфейс. Значение типа in_addr представляет собой упакованную двоичную строку, содержащую два 32-битных IP-адреса.
IP_MULTICAST_LOOP	0, 1	Разрешает или запрещает отправку копий пакетов на тот узел, откуда он был отправлен.
IP_MULTICAST_TTL	uint8	Определяет «время жизни» (time-to-live) исходящих многоадресных пакетов. Значение типа int8 представляет собой упакованную двоичную строку, содержащую 8-битное целое число без знака.
IP_OPTIONS	*ipopts*	Параметры IP-заголовка. Значение типа *ipopts* представляет собой упакованную двоичную строку длиной не более 44 байтов. Содержимое этой строки описывается в RFC 791.
IP_RECVDSTADDR	0, 1	Разрешает или запрещает принимать IP-адрес получателя в виде дейтаграммы.

(продолжение)

Имя параметра	Значение	Описание
IP_RECVOPTS	0, 1	Разрешает или запрещает принимать все IP-параметры в виде дейтаграммы.
IP_RECVRETOPTS	0, 1	Разрешает или запрещает принимать IP-параметры с ответом.
IP_RETOPTS	0, 1	То же, что и IP_RECVOPTS, но параметры остаются необработанными, с пустой отметкой времени и с пустой маршрутной записью.
IP_TOS	int	Тип обслуживания.
IP_TTL	int	Значение поля TTL пакета («время жизни»).

Следующие параметры доступны для уровня IPPROTO_IPV6:

Имя параметра	Значение	Описание
IPV6_CHECKSUM	0, 1	Разрешает или запрещает вычислять контрольную сумму.
IPV6_DONTFRAG	0, 1	Разрешает или запрещает фрагментировать пакеты, размер которых превышает размер MTU.
IPV6_DSTOPTS	*ip6_dest*	Параметры получателя. Значение типа *ip6_dest* представляет собой упакованную двоичную строку вида (*next, len, options*), где поле *next* содержит 8-битное целое число, определяющее тип параметра следующего заголовка; *len* – 8-битное целое число, определяющее длину заголовка в блоках по 8 байтов, не включая первые 8 байтов; и *options* – параметры кодирования.
IPV6_HOPLIMIT	int	Предельное количество транзитных узлов.
IPV6_HOPOPTS	*ip6_hbh*	Параметры транзитных узлов. Значение типа *ip6_hbh* интерпретируется так же, как и значение типа *ip6_dest*.
IPV6_JOIN_GROUP	*ip6_mreg*	Включение в группу многоадресной передачи. Значение типа *ip6_mreg* представляет собой упакованную двоичную строку вида (*multiaddr, index*), где *multiaddr* определяет 128-битный групповой адрес IPv6, а число *index* – 32-битный целочисленный индекс используемого локального интерфейса.
IPV6_LEAVE_GROUP	*ip6_mreg*	Выход из группы многоадресной передачи.
IPV6_MULTICAST_HOPS	int	Предельное количество транзитных узлов для многоадресных пакетов.

Имя параметра	Значение	Описание
IPV6_MULTICAST_IF	int	Индекс интерфейса для исходящих многоадресных пакетов.
IPV6_MULTICAST_LOOP	0, 1	Разрешает или запрещает отправку копий пакетов обратно локальному приложению.
IPV6_NEXTHOP	*sockaddr_in6*	Определяет адрес следующего транзитного узла для исходящих пакетов. Значение типа *sockaddr_in6* представляет собой упакованную двоичную строку, содержащую структуру sockaddr_in6 на языке C, определение которой обычно находится в файле <netinet/in.h>.
IPV6_PKTINFO	*ip6_pktinfo*	Структура с информацией о пакете. Значение типа *ip6_pktinfo* представляет собой упакованную двоичную строку вида (*addr, index*), где поле *addr* представляет 128-битный адрес IPv6, а поле *index* – 32-битный целочисленный индекс интерфейса.
IPV6_RECVDSTOPTS	0, 1	Разрешает или запрещает прием параметров получателя.
IPV6_RECVHOPLIMIT	0, 1	Разрешает или запрещает прием максимального количества транзитных узлов.
IPV6_RECVHOPOPTS	0, 1	Разрешает или запрещает прием параметров транзитных узлов.
IPV6_RECVPKTINFO	0, 1	Разрешает или запрещает прием информации о пакете.
IPV6_RECVRTHDR	0, 1	Разрешает или запрещает прием заголовка маршрутизации.
IPV6_RECVTCLASS	0, 1	Разрешает или запрещает принимать класс трафика.
IPV6_RTHDR	*ip6_rthdr*	Заголовок маршрутизации. Значение типа *ip6_rthdr* представляет собой упакованную двоичную строку вида (*next, len, type, segleft, data*), где поля *next*, *len*, *type* и *segleft* являются 8-битными целыми числами без знака, а поле *data* содержит информацию о маршрутизации. Подробности смотрите в RFC 2460.
IPV6_RTHDRDSTOPTS	*ip6_dest*	Заголовок с параметрами получателя, следующий перед заголовком с параметрами маршрутизации.
IPV6_RECVPATHMTU	0, 1	Разрешает или запрещает получение вспомогательных данных IPV6_PATHMTU.
IPV6_TCLASS	int	Класс трафика.
IPV6_UNICAST_HOPS	int	Предельное количество транзитных узлов для одноадресных пакетов.

(продолжение)

Имя параметра	Значение	Описание
IPV6_USE_MIN_MTU	-1, 0, 1	Разрешает или запрещает поиск маршрута с подходящим значением MTU. 1 – запрещает для всех получателей. -1 – запрещает только при многоадресной рассылке.
IPV6_V6ONLY	0, 1	Разрешает или запрещает устанавливать соединения только с узлами, поддерживающими IPv6.

Следующие параметры доступны для уровня SOL_TCP:

Имя параметра	Значение	Описание
TCP_CORK	0, 1	Значение 1 запрещает передавать неполные кадры.
TCP_DEFER_ACCEPT	0, 1	Активировать прослушивание сети только по прибытии данных в сокет.
TCP_INFO	*tcp_info*	Возвращает структуру с информацией о сокете. Содержимое структуры *tcp_info* зависит от реализации.
TCP_KEEPCNT	int	Максимальное количество попыток отправить служебное сообщение для поддержания соединения в активном состоянии, прежде чем признать соединение разорванным.
TCP_KEEPIDLE	int	Время простоя соединения в секундах, прежде чем начнется передача служебных сообщений, используемых для поддержания соединения в активном состоянии, если установлен параметр TCP_KEEPALIVE.
TCP_KEEPINTVL	int	Интервал в секундах между передачами служебных сообщений, используемых для поддержания соединения в активном состоянии.
TCP_LINGER2	int	Предельное время нахождения сокета в состоянии FIN_WAIT2.
TCP_MAXSEG	int	Максимальный размер сегмента для исходящих пакетов TCP.
TCP_NODELAY	0, 1	Значение 1 запрещает использование алгоритма Нагла.
TCP_QUICKACK	0, 1	При значении 1 пакеты ACK передаются немедленно. Запрещает использование алгоритма задержки отправки пакетов ACK.
TCP_SYNCNT	int	Число повторных попыток передачи пакета SYN до того, как будет принято решение о невозможности установить соединение.

Имя параметра	Значение	Описание
TCP_WINDOW_CLAMP	int	Определяет верхнюю границу предлагаемого размера окна TCP.

`s.gettimeout()`

Возвращает текущее значение предельного времени ожидания в секундах, если установлено. Возвращает число с плавающей точкой или None, если предельное время ожидания не было установлено.

s.ioctl(*control*, *option*)

Предоставляет ограниченный доступ к интерфейсу WSAIoctl в системе Windows. Для аргумента *control* поддерживается только значение SIO_RCVALL, которое применяется для перехвата всех IP-пакетов в сети. Для работы требует привилегий администратора. В аргументе *option* допускается передавать следующие значения:

Параметр	Описание
RCVALL_OFF	Запрещает прием всех пакетов IPv4 и IPv6.
RCVALL_ON	Включает режим прослушивания (promiscuous) и позволяет сокету принимать все пакеты IPv4 и IPv6, курсирующие в сети. Тип принимаемых пакетов определяется семейством адресов, установленных для сокета. Перехват пакетов других сетевых протоколов, таких как ARP, не выполняется.
RCVALL_IPLEVEL	Принимает все IP-пакеты, но отключает режим прослушивания. Будет захватывать все пакеты, направляемые на любой IP-адрес, присвоенный данному хосту.

s.listen(*backlog*)

Переводит сокет в режим ожидания входящих соединений. Аргумент *backlog* определяет максимальное количество запросов на соединение, ожидающих обработки, которые могут быть приняты, прежде чем новые запросы начнут отвергаться. Этот аргумент должен иметь значение не меньше 1, а значения 5 вполне достаточно для большинства применений.

s.makefile([*mode* [, *bufsize*]])

Создает объект файла, ассоциированного с сокетом. Аргументы *mode* и *bufsize* имеют тот же смысл, что и во встроенной функции open(). Объект файла использует дубликат файлового дескриптора сокета, созданного с помощью функции os.dup(), благодаря этому объект файла и объект сокета могут закрываться или утилизироваться сборщиком мусора независимо друг от друга. Для сокета *s* не должно быть определено предельное время ожидания и он не должен быть настроен на работу в неблокирующем режиме.

s.recv(*bufsize* [, *flags*])

Принимает данные из сокета. Данные возвращаются в виде строки. Максимальный объем принимаемых данных определяется аргументом *bufsize*.

В аргументе *flags* может передаваться дополнительная информация о сообщении, но обычно он не используется (в этом случае он по умолчанию принимает значение 0). Если используется, в нем, как правило, передается значение одной из следующих констант (фактические значения зависят от системы):

Константа	Описание
MSG_DONTROUTE	Не использовать таблицу маршрутизации (только для передачи).
MSG_DONTWAIT	Использовать неблокирующий режим.
MSG_EOR	Признак последнего сообщения в записи. Обычно используется только при передаче данных в сокеты типа SOCK_SEQPACKET.
MSG_PEEK	Просмотр данных без их уничтожения (только для приема).
MSG_OOB	Прием/передача внеочередных данных.
MSG_WAITALL	Не возвращать управление, пока не будет принято запрошенное количество байтов (только для приема).

s.recv_into(*buffer* [, *nbytes* [, *flags*]])

То же, что и recv(), за исключением того, что данные записываются в объект *buffer*, поддерживающий интерфейс буферов. В аргументе *nbytes* передается максимальное количество принимаемых байтов. Если этот аргумент отсутствует, максимальный размер определяется из размера буфера *buffer*. Аргумент *flags* имеет тот же смысл, что и в методе recv().

s.recvfrom(*bufsize* [, *flags*])

Действует аналогично методу recv(), но возвращает пару значений (*data, address*), где в поле *data* возвращается строка с принятыми данными, а в поле *address* возвращается адрес сокета, отправившего данные. Необязательный аргумент *flags* имеет тот же смысл, что и в методе recv(). Этот метод в первую очередь предназначен для работы с протоколом UDP.

s.recvfrom_info(*buffer* [, *nbytes* [, *flags*]])

То же, что и recvfrom(), но принимаемые данные сохраняются в объекте *buffer*. Аргумент *nbytes* определяет максимальное количество принимаемых байтов. Если этот аргумент отсутствует, максимальный размер определяется из размера буфера *buffer*. Аргумент *flags* имеет тот же смысл, что и в методе recv().

s.send(*string* [, *flags*])

Посылает данные через сетевое соединение. Необязательный аргумент *flags* имеет тот же смысл, что и в методе recv(). Возвращает **количество отправленных байтов**. Это число может оказаться меньше количества байтов в строке *string*. В случае ошибки возбуждает исключение.

s.sendall(*string* [, *flags*])

Посылает данные через сетевое соединение; при этом, прежде чем вернуть управление, пытается отправить все данные. В случае успеха возвращает

None; в случае ошибки возбуждает исключение. Аргумент *flags* имеет тот же смысл, что и в методе send().

s.sendto(*string* [, *flags*], *address*)

Посылает данные через сетевое соединение. Необязательный аргумент *flags* имеет тот же смысл, что и в методе recv(). В аргументе *address* передается кортеж вида (*host*, *port*), определяющий адрес удаленного хоста. К моменту вызова этого метода сокет не должен быть подключен. Возвращает количество отправленных байтов. Этот метод в первую очередь предназначен для работы с протоколом UDP.

s.setblocking(*flag*)

Если в аргументе *flag* передается ноль, сокет переводится в неблокирующий режим работы. В противном случае устанавливается блокирующий режим (по умолчанию). При работе в неблокирующем режиме, если вызов метода recv() не находит никаких данных или вызов метода send() не может немедленно отправить данные, возбуждается исключение socket.error. При работе в блокирующем режиме вызовы этих методов в подобных обстоятельствах блокируются, пока не появится возможность продолжить работу.

s.setsockopt(*level*, *optname*, *value*)

Устанавливает значение *value* для параметра *option* сокета. Аргументы *level* и *optname* имеют тот же смысл, что и в методе getsockopt(). В аргументе *value* может передаваться целое число или строка, представляющая содержимое буфера. В последнем случае приложение должно гарантировать, что строка содержит допустимое значение. Перечень допустимых имен параметров, их значений и дополнительное описание приводятся в описании метода getsockopt().

s.settimeout(*timeout*)

Устанавливает предельное время ожидания для операций, выполняемых сокетом. В аргументе *timeout* передается число с плавающей точкой, определяющее интервал времени в секундах. Значение None означает отсутствие предельного времени ожидания. По истечении указанного предельного времени ожидания операции будут возбуждать исключение socket.timeout. Вообще говоря, предельное время ожидания должно устанавливаться сразу же после создания сокета, так как оно также применяется к операциям, связанным с созданием логического соединения (таким как connect()).

s.shutdown(*how*)

Отключает одну или обе стороны соединения. Если в аргументе *how* передается значение 0, дальнейший прием данных будет запрещен. Если в аргументе *how* передается значение 1, будет запрещена дальнейшая передача данных. Если в аргументе *how* передается значение 2, будут запрещены и прием, и передача данных.

Помимо этих методов экземпляр *s* сокета обладает следующими свойствами, доступными только для чтения, которые соответствуют аргументам функции socket().

Свойство	Описание
s.family	Семейство адресов сокета (например, AF_INET)
s.proto	Протокол сокета
s.type	Тип сокета (например, SOCK_STREAM)

Исключения

Ниже приводится перечень исключений, объявленных в модуле socket.

error

Это исключение возбуждается в случае ошибок, связанных с сетевым соединением или с адресом. Значением исключения является кортеж (errno, mesg), содержащий информацию об ошибке, возвращаемую соответствующим системным вызовом. Наследует исключение IOError.

herror

Возбуждается в случае ошибок, связанных с адресом. Значением исключения является кортеж (*herrno*, *hmesg*) с числовым кодом ошибки и текстом сообщения. Наследует исключение error.

gaierror

Возбуждается в случае ошибок, связанных с адресом, возникающих в функциях getaddrinfo() и getnameinfo(). Значением исключения является кортеж (errno, mesg), где в поле *errno* возвращается числовой код ошибки, а в поле *mesg* – строка с текстом сообщения. В поле *errno* может возвращаться значение одной из следующих констант, объявленных в модуле socket:

Константа	Описание
EAI_ADDRFAMILY	Семейство адресов не поддерживается.
EAI_AGAIN	Временная ошибка при определении адреса по имени хоста.
EAI_BADFLAGS	Недопустимые флаги.
EAI_BADHINTS	Недопустимое рекомендуемое значение.
EAI_FAIL	Критическая ошибка при определении адреса по имени хоста.
EAI_FAMILY	Семейство адресов не поддерживается хостом.
EAI_MEMORY	Ошибка выделения памяти.
EAI_NODATA	Нет ни одного адреса, ассоциированного с именем узла.
EAI_NONAME	Имя узла или имя службы неизвестны.
EAI_PROTOCOL	Протокол не поддерживается.
EAI_SERVICE	Служба с указанным именем не поддерживается сокетом данного типа.
EAI_SOCKTYPE	Указанный тип сокетов не поддерживается.
EAI_SYSTEM	Системная ошибка.

`timeout`

Возбуждается по истечении предельного времени ожидания. Это исключение может возникать только в случае вызова функции `setdefaulttimeout()` или метода `settimeout()` объекта сокета. Значением исключения является строка `'timeout'`. Наследует исключение `error`.

Пример

Во вводной части главы приводился простой пример открытия соединения TCP. Ниже приводится пример реализации простого эхо-сервера, возвращающего полученные данные, использующего протокол UDP:

```
# Сервер сообщений, действующий по протоколу UDP
# Принимает небольшие пакеты и отправляет их обратно
import socket
s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
s.bind((“”,10000))
while True:
    data, address = s.recvfrom(256)
    print(“Получены данные с адреса %s” % str(address))
    s.sendto(b”echo:” + data, address)
```

Ниже приводится пример клиента, отправляющего сообщения серверу, реализация которого приводится выше:

```
# Клиент сообщений, действующий по протоколу UDP
import socket
s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
s.sendto(b”Hello World”, (“”, 10000))
resp, addr = s.recvfrom(256)
print(resp)
s.sendto(b”Spam”, (“”, 10000))
resp, addr = s.recvfrom(256)
print(resp)
s.close()
```

Примечания

- Не все константы и параметры сокетов доступны во всех платформах. Если перед вами стоит цель обеспечить переносимость программ, вы должны использовать только те параметры, которые описываются в источниках, таких как книга У. Ричарда Стивенса (W. Richard Stevens) «UNIX Network Programming»,[1] упоминавшаяся в начале этого раздела.
- Обратите внимание на отсутствие в модуле `socket` системных вызовов `recvmsg()` и `sendmsg()`, которые обычно используются для работы со вспомогательными данными и дополнительными сетевыми параметрами, имеющими отношение к заголовкам пакетов, маршрутизации и другим особенностям. Чтобы получить эту функциональность, можно восполь-

[1] Стивенс У. «UNIX. Разработка сетевых приложений». – Пер. с англ. – СПб.: Питер, 2003.

зоваться сторонними модулями, такими как PyXAPI (*http://pypi.python.org/pypi/PyXAPI*).

- Между операциями, выполняющимися в неблокирующем режиме, и операциями, предусматривающими выход по истечении предельного ожидания, существует важное отличие. Когда метод сокета выполняется в неблокирующем режиме, он возвращает управление немедленно, с признаком ошибки, если операция может быть заблокирована. Когда определено предельное время ожидания, методы возвращают признак ошибки, только если операция не была выполнена в течение указанного интервала времени.

Модуль ssl

Модуль `ssl` используется, чтобы обертывать объекты сокетов протоколом безопасных соединений (Secure Sockets Layer, SSL), который обеспечивает шифрование данных и аутентификацию. Реализация этого модуля опирается на использование библиотеки OpenSSL (*http://www.openssl.org*). Полное обсуждение теории и практики применения SSL выходит далеко за рамки этого раздела. Поэтому здесь будут рассмотрены только самые основные приемы использования этого модуля; при этом предполагается, что вы знакомы с настройкой SSL, с ключами, сертификатами и другими особенностями:

`wrap_socket(`*`sock`* `[, **`*`opts`*`])`

Обертывает существующий сокет *sock* (созданный средствами модуля `socket`) поддержкой SSL и возвращает экземпляр класса `SSLSocket`. Эта функция должна вызываться перед вызовом метода `connect()` или `accept()`. В аргументе *opts* передаются именованные аргументы, определяющие дополнительные параметры настройки.

Именованный аргумент	Описание
`server_side`	Логический флаг, который определяет, будет ли сокет выступать в роли сервера (`True`) или клиента (`False`). По умолчанию используется значение `False`.
`keyfile`	Файл ключа, используемый для идентификации локальной стороны соединения. Это должен быть файл в формате PEM и обычно этот параметр используется, если в параметре `certfile` указан файл, в котором отсутствует ключ.
`certfile`	Файл сертификата, используемый для идентификации локальной стороны соединения. Это должен быть файл в формате PEM.
`cert_reqs`	Определяет, должен ли файл сертификата запрашиваться у другой стороны соединения и должен ли он быть проверен. Значение `CERT_NONE` означает, что сертификат не требуется, `CERT_OPTIONAL` – сертификат не требуется, но при его получении он будет проверен, а значение `CERT_REQUIRED` означает, что сертификат обязателен и будет проверен. Если предполагается проверка сертификата, также должен быть установлен параметр `ca_certs`.

ca_certs	Имя файла, где хранится сертификат центра сертификации, который будет использоваться для проверки.
ssl_version	Используемая версия протокола SSL. Возможные значения: PROTOCOL_TLSv1, PROTOCOL_SSLv2, PROTOCOL_SSLv23 и PROTOCOL_SSLv3. По умолчанию используется версия протокола PROTOCOL_SSLv3.
do_handshake_on_connect	Логический флаг, который определяет, будет ли процедура подключения по протоколу SSL выполняться автоматически. По умолчанию используется значение True.
suppress_ragged_eofs	Определяет, как функция read() будет обрабатывать неожиданное получение признака конца файла. Если указано значение True (по умолчанию), будет возвращаться признак конца файла. Если указано значение False, будет возбуждаться исключение.

Экземпляр *s* класса SSLSocket, наследующего класс socket.socket, поддерживает следующие операции:

s.cipher()

Возвращает кортеж (*name*, *version*, *secretbits*), где в поле *name* возвращается название используемого алгоритма шифрования, в поле *version* возвращается версия протокола SSL и в поле *secretbits* – длина используемого ключа шифрования в битах.

s.do_handshake()

Выполняет начальный этап соединения SSL. Обычно этот метод вызывается автоматически, если только функции wrap_socket() не был передан именованный аргумент do_handshake_on_connect со значением False. Если для самого сокета *s* был установлен неблокирующий режим работы, возбуждает исключение SSLError в случае невозможности завершить операцию. В атрибуте e.args[0] экземпляра исключения SSLError будет возвращено значение SSL_ERROR_WANT_READ или SSL_ERROR_WANT_WRITE, в зависимости от того, какая операция должна быть выполнена. Чтобы продолжить начальную процедуру подключения после того, как операция чтения или записи сможет быть выполнена, достаточно просто еще раз вызвать метод *s*.do_handshake().

s.getpeercert([*binary_form*])

Возвращает сертификат другого конца соединения, если таковой имеется. В случае отсутствия сертификата возвращает None. Если сертификат имеется, но он не прошел проверку, возвращается пустой словарь. Если был принят сертификат и проверка его увенчалась успехом, возвращается словарь с ключами 'subject' и 'notAfter'. Если в аргументе *binary_form* передается значение True, сертификат возвращается в виде последовательности байтов в формате DER.

s.read([*nbytes*])

Читает до *nbytes* байтов данных и возвращает их. При вызове без аргумента *nbytes* читает до 1024 байтов.

s.write(*data*)

Записывает строку байтов *data*. Возвращает количество фактически записанных байтов.

s.unwrap()

Закрывает соединение SSL и возвращает объект сокета, который далее может использоваться для обмена незашифрованными данными.

Ниже перечислены вспомогательные функции, объявленные в модуле:

cert_time_to_seconds(*timestring*)

Преобразует строку *timestring* из формата, используемого в сертификатах, в число с плавающей точкой, совместимое с функцией time.time().

DER_cert_to_PEM_cert(*derbytes*)

Принимает строку байтов *derbytes* с сертификатом в формате DER и возвращает строку с версией сертификата в формате PEM.

PEM_cert_to_DER_cert(*pemstring*)

Принимает строку *pemstring* с сертификатом в формате PEM и возвращает строку байтов с версией сертификата в формате DER.

get_server_certificate(*addr* [, *ssl_version* [, *ca_certs*]])

Извлекает сертификат сервера SSL и возвращает в виде строки в формате PEM. В аргументе *addr* передается адрес сервера в виде (*hostname*, *port*). В аргументе *ssl_version* передается номер версии протокола SSL, а в аргументе *ca_certs* – имя файла с сертификатом центра сертификации, как описывалось в описании функции wrap_socket().

RAND_status()

Возвращает True, если с точки зрения SSL генератор псевдослучайных чисел обеспечивает достаточный уровень случайности.

RAND_egd(*path*)

Читает 256 случайных байтов из демона энтропии и добавляет их в генератор псевдослучайных чисел. Аргумент *path* определяет имя сокета демона.

RAND_add(*bytes*, *entropy*)

Добавляет байты из строки байтов *bytes* в генератор псевдослучайных чисел. В аргументе *entropy* передается неотрицательное число с плавающей точкой, определяющее нижнюю границу энтропии.

Примеры

Следующий пример демонстрирует, как с помощью этого модуля можно открыть SSL-соединение со стороны клиента:

```
import socket, ssl

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
ssl_s = ssl.wrap_socket(s)
ssl_s.connect((‘gmail.google.com’,443))
print(ssl_s.cipher())

# Отправить запрос
ssl_s.write(b"GET / HTTP/1.0\r\n\r\n")
```

```
# Получить ответ
while True:
    data = ssl_s.read()
    if not data: break
    print(data)
ssl_s.close()
```

Ниже приводится пример сервера времени, действующего по протоколу SSL:

```
import socket, ssl, time

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR,1)
s.bind((‘’,12345))
s.listen(5)

while True:
    client, addr = s.accept() # Принять соединение
    print “Получен запрос на соединение с “, addr
    client_ssl = ssl.wrap_socket(client,
                                 server_side=True,
                                 certfile=”timecert.pem”)

    client_ssl.sendall(b”HTTP/1.0 200 OK\r\n”)
    client_ssl.sendall(b”Connection: Close\r\n”)
    client_ssl.sendall(b”Content-type: text/plain\r\n\r\n”)
    resp = time.ctime() + “\r\n”
    client_ssl.sendall(resp.encode(‘latin-1’))
    client_ssl.close()
    client.close()
```

Чтобы запустить этот сервер, необходимо сохранить подписанный сертификат в файле `timecert.pem`. Этот сертификат можно создать с помощью команды UNIX:

```
% openssl req -new -x509 -days 30 -nodes -out timecert.pem -keyout timecert.pem
```

Чтобы протестировать работу этого сервера, попробуйте соединиться с ним с помощью веб-броузера, указав адрес URL `‘https://localhost:1234’`. Если все работает, броузер выведет предупреждение о том, что используется самоподписанный сертификат. После того как вы согласитесь продолжить, в окне броузера появится вывод, полученный от сервера.

Модуль SocketServer

В Python 3 этот модуль называется `socketserver`. Модуль `SocketServer` объявляет классы, упрощающие реализацию серверов на основе сокетов TCP, UDP и домена UNIX.

Обработчики

Для использования модуля необходимо объявить класс обработчика, производный от базового класса `BaseRequestHandler`. Экземпляр *h* класса `Base-`

RequestHandler реализует один или более методов из тех, что перечислены ниже:

h.finish()

Вызывается для выполнения завершающих операций после того, как метод handle() закончит работу. По умолчанию этот метод ничего не делает. Он не вызывается, если метод setup() или handle() возбуждает исключение.

h.handle()

Этот метод выполняет фактическую работу в соответствии с запросом. Он вызывается без аргументов, но может использовать некоторые атрибуты экземпляра для получения необходимой информации. Атрибут *h*.request содержит запрос, *h*.client_address – адрес клиента и *h*.server – экземпляр сервера, вызвавшего обработчик. Для потоков, таких как TCP, атрибут *h*.request будет содержать объект сокета. Для дейтаграмм он будет содержать строку байтов с принятыми данными.

h.setup()

Этот метод вызывается перед методом handle() для выполнения операций по инициализации. По умолчанию этот метод ничего не делает. Если необходимо реализовать в сервере дополнительные настройки соединения, такие как подключение по протоколу SSL, эти операции должны быть реализованы в этом методе.

Ниже приводится пример класса обработчика, реализующего простой сервер времени, который может работать как с потоками, так и с дейтаграммами:

```
try:
    from socketserver import BaseRequestHandler # Python 3
except ImportError:
    from SocketServer import BaseRequestHandler # Python 2
import socket
import time

class TimeServer(BaseRequestHandler):
    def handle(self):
        resp = time.ctime() + "\r\n"
        if isinstance(self.request,socket.socket):
            # Работа с потоком
            self.request.sendall(resp.encode('latin-1'))
        else:
            # Работа с дейтаграммой
            self.server.socket.sendto(resp.encode('latin-1'),
                                      self.client_address)
```

Если заранее известно, что обработчик будет работать только с потоковыми протоколами, такими как TCP, в качестве родительского можно использовать класс StreamRequestHandler, а не BaseRequestHandler. Этот класс определяет два атрибута: *h*.wfile – объект, похожий на файл, который отправляет данные клиенту, и *h*.rfile – объект, похожий на файл, позволяющий принимать данные от клиента. Например:

```
try:
    from socketserver import StreamRequestHandler # Python 3
except ImportError:
    from SocketServer import StreamRequestHandler # Python 2
import time

class TimeServer(StreamRequestHandler):
    def handle(self):
        resp = time.ctime() + "\r\n"
        self.wfile.write(resp.encode('latin-1'))
```

Если обработчик должен работать только с отдельными пакетами и всегда возвращать ответ отправителю, в качестве родительского можно использовать класс `DatagramRequestHandler` вместо `BaseRequestHandler`. Этот класс реализует тот же интерфейс файлов, что и класс `StreamRequestHandler`. Например:

```
try:
    from socketserver import DatagramRequestHandler # Python 3
except ImportError:
    from SocketServer import DatagramRequestHandler # Python 2
import time

class TimeServer(DatagramRequestHandler):
    def handle(self):
        resp = time.ctime() + "\r\n"
        self.wfile.write(resp.encode('latin-1')
```

В данном случае все данные, записанные в `self.wfile`, собираются в единый пакет, который отправляется клиенту по завершении работы метода `handle()`.

Серверы

Чтобы задействовать обработчик, его необходимо подключить к объекту сервера. В модуле объявлено четыре основных класса серверов:

`TCPServer(`*`address`*`, `*`handler`*`)`

Сервер, поддерживающий протокол TCP IPv4. В аргументе *address* передается кортеж вида (*host*, *port*). В аргументе *handler* – экземпляр подкласса класса `BaseRequestHandler`, описанного выше.

`UDPServer(`*`address`*`, `*`handler`*`)`

Сервер, поддерживающий протокол UDP IPv4. Аргументы *address* и *handler* имеют тот же смысл, что и в конструкторе `TCPServer()`.

`UnixStreamServer(`*`address`*`, `*`handler`*`)`

Сервер, реализующий потоковый протокол с использованием сокетов домена UNIX. Наследует класс `TCPServer`.

`UnixDatagramServer(`*`address`*`, `*`handler`*`)`

Сервер, поддерживающий протокол дейтаграмм с использованием сокетов домена UNIX. Наследует класс `UDPServer`.

Экземпляры всех четырех классов серверов обладают следующими основными методами:

s.fileno()

Возвращает целочисленный дескриптор файла серверного сокета. Наличие этого метода обеспечивает возможность использовать экземпляры серверов в операциях опроса, таких как функция select().

s.serve_forever()

Обслуживает неограниченное число запросов.

s.shutdown()

Останавливает цикл serve_forever().

Следующие атрибуты позволяют получать некоторую основную информацию о настройках действующего сервера:

s.RequestHandlerClass

Пользовательский класс обработчика, который был передан конструктору сервера.

s.server_address

Адрес, на котором сервер ожидает получения запросов от клиентов, например такой кортеж: ('127.0.0.1', 80).

s.socket

Объект сокета, используемый для приема входящих запросов.

Ниже приводится пример использования экземпляра класса TimeHandler в составе сервера TCP:

```
from SocketServer import TCPServer

serv = TCPServer(('',10000,TimeHandler)
serv.serve_forever()
```

Ниже приводится пример использования обработчика в составе сервера UDP:

```
from SocketServer import UDPServer

serv = UDPServer(('',10000,TimeHandler)
serv.serve_forever()
```

Ключевым аспектом модуля SocketServer является обособленность обработчиков от серверов. То есть один и тот же обработчик можно подключить к серверам самых разных типов, не меняя его реализацию.

Определение собственных серверов

Зачастую для серверов требуется определять различные параметры настройки, чтобы учесть такие особенности, как различные семейства адресов, предельное время ожидания, многозадачность и другие. Для этого необходимо определить свой класс, производный от одного из четырех базо-

вых классов серверов, описанных в предыдущем разделе. Для настройки параметров сокета, составляющего основу сервера, могут быть определены следующие атрибуты класса:

Server.address_family

Семейство адресов, используемое сокетом сервера. По умолчанию используется значение socket.AF_INET. Если необходимо обеспечить поддержку IPv6, следует использовать значение socket.AF_INET6.

Server.allow_reuse_address

Логический флаг, разрешающий или запрещающий повторное использование адреса сокета. Это бывает удобно, когда необходимо быстро перезапустить сервер на том же порту, после того как программа завершится (в противном случае придется ожидать несколько минут). По умолчанию используется значение False.

Server.request_queue_size

Размер очереди запросов, который передается методу listen() сокета. По умолчанию используется значение 5.

Server.socket_type

Тип сокета, используемого сервером, такой как socket.SOCK_STREAM или socket.SOCK_DGRAM.

Server.timeout

Предельное время ожидания в секундах, в течение которого сервер будет ожидать поступления новых запросов. По истечении этого интервала времени будет вызываться метод handle_timeout() сервера (описывается ниже), после чего сервер опять возвращается к ожиданию. Это значение *не используется* для установки предельного времени ожидания в сокете. Однако если для сокета было определено предельное время ожидания, вместо этого значения будет использоваться значение из сокета.

Ниже приводится пример создания сервера, который позволяет повторно использовать номер порта:

```
from SocketServer import TCPServer

class TimeServer(TCPServer):
    allow_reuse_address = True

serv = TimeServer(('',10000,TimeHandler)
serv.serve_forever()
```

При желании в классах, производных от базовых классов серверов, можно переопределять следующие методы. Если вы будете переопределять какой-либо из этих методов, не забудьте вызвать одноименный метод суперкласса.

Server.activate()

Выполняет операцию listen() на стороне сервера. Сокет сервера доступен в виде атрибута self.socket.

Server.bind()

Выполняет операцию bind() на стороне сервера.

Server.handle_error(*request*, *client_address*)

Обрабатывает неперехваченные исключения, которые возникают в процессе работы. Для получения информации о последнем исключении следует использовать функцию sys.exc_info() или функции из модуля traceback.

Server.handle_timeout()

Обрабатывает ситуации, когда операции завершаются по истечении предельного времени ожидания. Переопределяя этот метод и изменяя значение предельного времени ожидания, можно в цикл событий сервера интегрировать дополнительные операции.

Server.verify_request(*request*, *client_address*)

Этот метод можно переопределить, чтобы реализовать проверку соединения перед обработкой запроса. Это может быть реализация сетевой защиты или каких-то других проверок.

Наконец, дополнительные возможности сервера можно получить за счет использования классов-примесей. С их помощью может быть добавлена многозадачность, реализованная на основе потоков управления или дочерних процессов. Для этой цели определены следующие классы:

ForkingMixIn

Класс-примесь, который в UNIX создает дочерние процессы сервера, позволяя одновременно обслуживать множество клиентов. Атрибут класса max_children определяет максимальное количество дочерних процессов, а атрибут класса timeout определяет интервал времени, через который будут выполняться попытки ликвидировать процессы-зомби. Атрибут экземпляра active_children содержит количество активных процессов.

ThreadingMixIn

Класс-примесь, который модифицирует сервер так, что он создает новые потоки управления, позволяя одновременно обслуживать множество клиентов. Этот класс не имеет ограничений на количество создаваемых потоков управления. По умолчанию создаются недемонические потоки, если в атрибут класса daemon_threads не записать значение True.

Чтобы добавить к серверу эти возможности, следует использовать механизм множественного наследования и класс-примесь указывать первым в списке. Например, ниже приводится пример сервера времени, запускающего дочерние процессы:

```
from SocketServer import TCPServer, ForkingMixIn

class TimeServer(ForkingMixIn, TCPServer):
    allow_reuse_address = True
    max_children = 10

serv = TimeServer(('',10000),TimeHandler)
serv.serve_forever()
```

Поскольку ситуация, когда параллельно выполняется несколько серверов, достаточно типична, для этой ситуации в модуле SocketServer предопределены следующие классы серверов.

- ForkingUDPServer(*address*, *handler*)
- ForkingTCPServer(*address*, *handler*)
- ThreadingUDPServer(*address*, *handler*)
- ThreadingTCPServer(*address*, *handler*)

В действительности эти классы объявлены, как производные классы от классов-примесей и классов серверов. В качестве примера ниже приводится объявление класса ForkingTCPServer:

```
class ForkingTCPServer(ForkingMixIn, TCPServer): pass
```

Создание собственных серверов приложений

Класс SocketServer часто используется другими модулями из стандартной библиотеки для реализации серверов, работающих с прикладными протоколами, такими как HTTP и XML-RPC. Функциональность этих серверов также можно приспосабливать под свои нужды через множественное наследование и переопределение методов, объявленных в базовых классах. Например, ниже приводится сервер XML-RPC, порождающий дочерние процессы, который принимает только соединения, исходящие с петлевого (loopback) интерфейса:

```
try:
    from xmlrpc.server import SimpleXMLRPCServer       # Python 3
    from socketserver import ForkingMixIn
except ImportError:                                    # Python 2
    from SimpleXMLRPCServer import SimpleXMLRPCServer
    from SocketServer import ForkingMixIn

class MyXMLRPCServer(ForkingMixIn,SimpleXMLRPCServer):
    def verify_request(self, request, client_address):
        host, port = client_address
        if host != '127.0.0.1':
            return False
        return SimpleXMLRPCServer.verify_request(self,request,client_address)

# Пример использования
def add(x,y):
    return x+y
server = MyXMLRPCServer(("",45000))
server.register_function(add)
server.serve_forever()
```

Чтобы опробовать этот пример, необходимо импортировать модуль xmlrpclib. Запустите сервер, реализация которого представлена выше, а затем запустите отдельный процесс интерпретатора Python:

```
>>> import xmlrpclib
>>> s = xmlrpclib.ServerProxy("http://localhost:45000")
```

```
>>> s.add(3,4)
7
>>>
```

Чтобы убедиться, что сервер отвергает попытки соединения с другого адреса, попробуйте выполнить тот же код на другом компьютере в сети. Для этого замените строку "localhost" сетевым именем компьютера, на котором запущен сервер.

22

Разработка интернет-приложений

В этой главе описываются модули, обеспечивающие поддержку прикладных протоколов Интернета, включая HTTP, XML-RPC, FTP and SMTP. Темы, связанные с веб-программированием, такие как разработка CGI-сценариев, рассматриваются в главе 23 «Веб-программирование». Модули, связанные с обработкой форматов данных, широко используемых в Интернете, описываются в главе 24 «Обработка и кодирование данных в Интернете».

Организация сетевых библиотечных модулей является одной из областей, где между Python 2 и 3 имеются существенные различия. В этой главе за основу взята организация модулей, принятая в **Python 3, как более логичная**. При этом функциональные возможности модулей в этих двух версиях **Python к моменту написания этих строк были практически идентичны**. В случае необходимости в разделах будут указываться имена модулей Python 2.

Модуль ftplib

Модуль `ftplib` реализует клиентскую часть протокола **FTP**. На практике достаточно редко приходится использовать этот модуль непосредственно, потому что имеется пакет `urllib`, который реализует более высокоуровневый интерфейс. Однако этот модуль может пригодиться, когда в программе потребуется иметь более полный контроль над низкоуровневыми особенностями **FTP-соединения. Для работы с этим модулем совсем не лишним** будет поближе познакомиться с протоколом **FTP, который описывается** в RFC 959.

Модуль определяет единственный класс, используемый для создания FTP-соединений:

`FTP([`*`host`*` [, `*`user`*` [, `*`passwd`*` [, `*`acct`*` [, `*`timeout`*`]]]]])`

Создает объект, представляющий FTP-соединение. В аргументе *`host`* передается строка с именем хоста. Аргументы *`user`*, *`passwd`* и *`acct`* являются не-

обязательными и определяют имя пользователя, пароль и учетную запись. При вызове без аргументов, чтобы установить фактическое соединение, потребуется явно вызвать методы `connect()` и `login()`. Если указать аргумент *host*, автоматически будет вызван метод `connect()`. Если указать аргументы *user*, *passwd* и *acct*, автоматически будет вызван метод `login()`. Аргумент *timeout* определяет предельное время ожидания в секундах.

Экземпляр *f* класса `FTP` обладает следующими методами:

f.`abort()`

Выполняет попытку прервать продолжающуюся передачу файла. Вызов этого метода может не давать желаемого результата, в зависимости от особенностей удаленного сервера.

f.`close()`

Закрывает FTP-соединение. После вызова этого метода никакие другие операции не должны выполняться над объектом *f*.

f.`connect(`*host* `[, ` *port* `[, ` *timeout*`]])`

Открывает FTP-соединение с указанным именем хоста и номером порта. В аргументе *host* передается строка с именем хоста, а в аргументе *port* – целочисленный номер порта сервера FTP (по умолчанию используется порт 21). Аргумент *timeout* определяет предельное время ожидания в секундах. Нет необходимости вызывать этот метод, если имя хоста уже передавалось конструктору `FTP()`.

f.`cwd(`*pathname*`)`

Изменяет текущий рабочий каталог на сервере, выполняя переход по указанному пути *pathname*.

f.`delete(`*filename*`)`

Удаляет файл *filename* на сервере.

f.`dir([`*dirname* `[, ... [, ` *callback*`]]])`

Создает список содержимого каталога, который создается командой `'LIST'`. В необязательном аргументе *dirname* можно указать имя каталога, для которого должен быть составлен список содержимого.[1] Кроме того, все дополнительные аргументы метода будут просто передаваться команде `'LIST'`. Если в последнем аргументе *callback* передать функцию, она будет использована как функция обратного вызова для обработки возвращаемого списка с содержимым каталога. Эта функция используется точно так же, как и в методе `retrlines()`. По умолчанию метод выводит содержимое списка в поток `sys.stdout`.

f.`login([`*user*`, [`*passwd* `[, ` *acct*`]]])`

Выполняет регистрацию на сервере, используя указанные имя пользователя, пароль и учетную запись. В аргументе *user* передается строка с именем пользователя; аргумент по умолчанию имеет значение `'anonymous'`. В аргу-

[1] По умолчанию используется текущий рабочий каталог на сервере. – *Прим. перев.*

менте *passwd* передается строка с паролем; аргумент по умолчанию имеет значение '' (пустая строка). В аргументе *acct* передается строка, по умолчанию – пустая строка. Нет необходимости вызывать этот метод, если эта информация уже передавалась конструктору FTP().

f.mkd(pathname)

Создает новый каталог на сервере.

f.ntransfercmd(*command* [, *rest*])

То же, что и transfercmd(), за исключением того, что этот метод возвращает кортеж (*sock*, *size*), где в поле *sock* возвращается объект сокета, соответствующий соединению, через которое идет обмен данными, а в поле *size* возвращается ожидаемый объем данных в байтах или None, если объем заранее неизвестен.

f.pwd()

Возвращает строку с именем текущего рабочего каталога на сервере.

f.quit()

Закрывает FTP-соединение, отправляя серверу команду 'QUIT'.

f.rename(*oldname*, *newname*)

Переименовывает файл на сервере.

f.retrbinary(*command*, *callback* [, *blocksize* [, *rest*]])

Возвращает результат выполнения команды на сервере, используя двоичный режим передачи. В аргументе *command* передается строка, определяющая соответствующую команду извлечения файла, которая практически всегда имеет вид 'RETR *filename*'. В аргументе *callback* передается функция обратного вызова, которая будет вызываться всякий раз при получении очередного блока данных. Эта функция будет вызываться с единственным аргументом, содержащим принятые данные в виде строки. Аргумент *blocksize* определяет максимальный размер блока. По умолчанию передача ведется блоками по 8192 байта. Необязательный аргумент *rest* определяет смещение относительно начала файла. Если указан, определяет позицию в файле, откуда должна начаться передача данных. Однако не все серверы FTP поддерживают такую возможность, поэтому использование этого аргумента может вызывать исключение error_reply.

f.retrlines(*command* [, *callback*])

Возвращает результат выполнения команды на сервере, используя текстовый режим передачи. В аргументе *command* передается строка, определяющая соответствующую команду, которая обычно имеет вид 'RETR *filename*'. В аргументе *callback* передается функция обратного вызова, которая будет вызываться всякий раз при получении очередной строки данных. Эта функция будет вызываться с единственным аргументом, содержащим принятые данные в виде строки. При вызове без аргумента *callback* возвращаемые данные будут выводиться в поток sys.stdout.

f.rmd(*pathname*)

Удаляет каталог *pathname* на сервере.

f.sendcmd(*command*)

Отправляет серверу простую команду и возвращает ответ. В аргументе *command* передается строка с командой. Этот метод должен использоваться только для выполнения команд, которые не связаны с передачей данных.

f.set_pasv(*pasv*)

Устанавливает пассивный режим. В аргументе *pasv* передается логический флаг, значение True в котором включает пассивный режим, а значение False — отключает. По умолчанию пассивный режим включен.

f.size(*filename*)

Возвращает размер файла *filename* в байтах. Возвращает None, если по каким-либо причинам размер файла определить невозможно.

f.storbinary(*command*, *file* [, *blocksize*])

Выполняет команду на сервере и возвращает результаты, используя двоичный режим передачи. В аргументе *command* передается строка с низкоуровневой командой, которая практически всегда имеет вид 'STOR *filename*', где подстрока *filename* определяет имя файла, который требуется сохранить на сервере. В аргументе *file* передается открытый объект файла, откуда вызовом метода *file*.read(*blocksize*) будут читаться данные и отправляться на сервер. Аргумент *blocksize* определяет размер блока при передаче. По умолчанию передача ведется блоками по 8192 байтов.

f.storlines(*command*, *file*)

Выполняет команду на сервере и возвращает результаты, используя текстовый режим передачи. В аргументе *command* передается строка с низкоуровневой командой, которая обычно имеет вид 'STOR *filename*'. В аргументе *file* передается открытый объект файла, откуда вызовом метода *file*.readline() будут читаться данные и отправляться на сервер.

f.transfercmd(*command* [, *rest*])

Инициирует передачу через **FTP-соединение обмена данными. Если** используется активный режим, отправляет команду 'PORT' или 'EPRT' и принимает соединение со стороны сервера. Если используется пассивный режим, отправляет команду 'EPSV' или 'PASV' с последующим запросом на соединение с сервером. В любом случае, как только соединение будет установлено, выполняется FTP-команда *command*. Этот метод возвращает объект сокета, соответствующего открытому соединению, предназначенному для передачи данных. Необязательный аргумент *rest* определяет смещение в байтах от начала запрашиваемого файла. Однако не все серверы **FTP под**держивают такую возможность, поэтому использование этого аргумента может вызывать исключение error_reply.

Пример

Следующий пример демонстрирует, как с помощью этого модуля выгрузить файл на сервер FTP:

```
host     = "ftp.foo.com"
username = "dave"
```

```
password = "1235"
filename = "somefile.dat"

import ftplib
ftp_serv = ftplib.FTP(host,username,password)
# Открыть файл, предназначенный для передачи
f = open(filename,"rb")
# Передать файл на сервер FTP
resp = ftp_serv.storbinary("STOR "+filename, f)
# Закрыть соединение
ftp_serv.close
```

Для загрузки документов с сервера FTP можно использовать пакет `urllib`. Например:

```
try:
    from urllib.request import urlopen   # Python 3
except ImportError:
    from urllib2 import urlopen          # Python 2

u = urlopen("ftp://username:password@somehostname/somefile")
contents = u.read()
```

Пакет http

Пакет `http` состоит из модулей, предназначенных для разработки клиентов и серверов HTTP, а также для управления механизмом сохранения состояния (cookies). Протокол передачи гипертекста (Hypertext Transfer Protocol, HTTP) – это простой текстовый протокол, который действует следующим образом:

1. Клиент устанавливает соединение с сервером HTTP и отправляет запрос с заголовком следующего вида:

```
GET /document.html HTTP/1.0
Connection: Keep-Alive
User-Agent: Mozilla/4.61 [en] (X11; U; SunOS 5.6 sun4u)
Host: rustler.cs.uchicago.edu:8000
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, image/png, */*
Accept-Encoding: gzip
Accept-Language: en
Accept-Charset: iso-8859-1,*,utf-8

Дополнительные данные
...
```

Первая строка определяет тип запроса, путь к документу (селектор) и версию протокола. Далее следует последовательность строк-заголовков с различной информацией о клиенте, например пароли, cookies, предпочтительные настройки механизма кэширования и версия клиентского программного обеспечения. Вслед за строками-заголовками следует одна пустая строка, которая указывает на окончание последовательности заголовков. Далее могут следовать дополнительные данные, когда вместе с запросом отправляются данные формы или производится вы-

грузка файла. Каждая строка заголовка должна завершаться символом возврата каретки и символом перевода строки (‘\r\n’).

2. Сервер возвращает ответ следующего вида:

```
HTTP/1.0 200 OK
Content-type: text/html
Content-length: 72883 bytes
...
Заголовок: данные

Данные
...
```

Первая строка ответа сервера определяет версию протокола HTTP, код успеха и возвращаемое сообщение. Далее следует последовательность строк-заголовков с информацией о типе возвращаемого документа, о размере документа, о версии программного обеспечения веб-сервера, cookies и так далее. Последовательность заголовков заканчивается пустой строкой, за которой следуют данные, составляющие тело запрошенного документа.

Наиболее часто используются следующие методы запросов:

Метод	Описание
GET	Получить документ.
POST	Послать данные формы.
HEAD	Вернуть только заголовки.
PUT	Выгрузить данные на сервер.

Коды ответов, которые наиболее часто возвращаются серверами, приводятся в табл. 22.1. В столбце «Символическая константа» приводится имя предопределенной переменной в модуле http.client, хранящей целочисленный код ответа, которая может использоваться для повышения удобочитаемости программного кода.

Таблица 22.1. Коды ответов, наиболее часто возвращаемые серверами

Код	Описание	Символическая константа
Коды успешного выполнения запроса		
200	Выполнено	OK
201	Создано	CREATED
202	Принято	ACCEPTED
204	Нет содержимого	NO_CONTENT

Код	Описание	Символическая константа
Переадресация (3xx)		
300	Несколько документов	MULTIPLE_CHOICES
301	Документ был перемещен по другому адресу	MOVED_PERMANENTLY
302	Документ временно перемещен по другому адресу	MOVED_TEMPORARILY
303	Документ не изменялся	NOT_MODIFIED
Ошибка клиента (4xx)		
400	Недопустимый запрос	BAD_REQUEST
401	Требуется авторизация	UNAUTHORIZED
403	Доступ к документу запрещен	FORBIDDEN
404	Документ не найден	NOT_FOUND
Ошибка сервера (5xx)		
500	Внутренняя ошибка сервера	INTERNAL_SERVER_ERROR
501	Не реализовано	NOT_IMPLEMENTED
502	Ошибка шлюза	BAD_GATEWAY
503	Служба недоступна	SERVICE_UNAVAILABLE

Заголовки, которые могут присутствовать как в запросах, так и в ответах, следуют формату, широко известному как **RFC-822. В общем случае каждый заголовок имеет вид** *Имязаголовка: данные*; дополнительные подробности вы найдете в документе **RFC. На практике практически никогда не приходится анализировать заголовки, так как в случае необходимости это делается автоматически.**

Модуль http.client (httplib)

Модуль `http.client` реализует низкоуровневую поддержку протокола HTTP со стороны клиента. В Python 2 этот модуль называется `httplib`. Вместо этого модуля предпочтительнее пользоваться функциями из пакета `urllib`. Модуль поддерживает обе версии протокола, **HTTP/1.0 и HTTP/1.1, и дополнительно позволяет использовать защищенные SSL-соединения, если интерпретатор Python был скомпилирован с поддержкой OpenSSL. На практике достаточно редко приходится использовать этот пакет непосредственно; вместо него лучше использовать пакет** `urllib`. Однако, учитывая большую важность протокола HTTP, можно столкнуться с ситуациями, когда потребуется решать задачи на более низком уровне, которые трудно или невозможно решить с помощью пакета `urllib`, например реализовать отправку запросов с командами, отличными от `GET` и `POST`. За дополнительной информацией о протоколе HTTP обращайтесь к RFC 2616 (HTTP/1.1) и RFC 1945 (HTTP/1.0).

Ниже перечислены классы, которые могут использоваться для создания HTTP-соединений с сервером:

`HTTPConnection(`*`host`* `[,`*`port`*`])`

Создает HTTP-соединение. В аргументе *host* передается имя хоста, а в аргументе *port* — номер порта. По умолчанию используется порт с номером 80. Возвращает экземпляр класса `HTTPConnection`.

`HTTPSConnection(`*`host`* `[, `*`port`* `[, key_file=`*`kfile`* `[, cert_file=`*`cfile`*`]]])`

Создает HTTP-соединение, используя защищенный сокет. По умолчанию используется порт с номером 443. Необязательные именованные аргументы `key_file` и `cert_file` определяют файлы частного ключа и сертификата клиента в формате **PEM, если они необходимы для аутентификации клиента**. Однако никакой проверки достоверности сертификата сервера не выполняется. Возвращает экземпляр класса `HTTPSConnection`.

Экземпляр *h* класса `HTTPConnection` или `HTTPSConnection` поддерживает следующие методы:

h`.connect()`

Инициирует соединение с хостом и портом, указанными в вызове конструктора `HTTPConnection()` или `HTTPSConnection()`. Другие методы вызывают его автоматически, если соединение еще не было установлено.

h`.close()`

Закрывает соединение.

h`.send(`*`bytes`*`)`

Отправляет строку байтов *bytes* серверу. Не рекомендуется пользоваться этим методом непосредственно, потому что это может привести к нарушениям в работе протокола запрос/ответ. Обычно он используется для передачи данных на сервер после вызова метода *h*`.endheaders()`.

h`.putrequest(`*`method`*`, `*`selector`* `[, `*`skip_host`* `[, `*`skip_accept_encoding`*`]])`

Отправляет запрос серверу. В аргументе *method* указывается метод HTTP, такой как `'GET'` или `'POST'`. Аргумент *selector* определяет возвращаемый объект, например: `'/index.html'`. В аргументах *skip_host* и *skip_accept_encoding* передаются флаги, запрещающие отправку заголовков HTTP `Host:` и `Accept-Encoding:`. По умолчанию оба аргумента получают значение `False`. Протокол HTTP/1.1 позволяет отправлять несколько запросов через одно и то же соединение, поэтому если соединение находится в состоянии, препятствующем отправке нового запроса, этот метод возбуждает исключение `CannotSendRequest`.

h`.putheader(`*`header`*`, `*`value`*`, ...)`

Отправляет серверу заголовок, соответствующий формату **RFC-822. Серверу** отправляется строка, состоящая из имени заголовка, двоеточия, пробела и значения. Дополнительные аргументы интерпретируются как дополнительные строки текста в заголовке. Если соединение находится в состоянии, препятствующем отправке заголовка, возбуждается исключение `CannotSendRequest`.

h.endheaders()

Отправляет серверу пустую строку, обозначающую конец последовательности заголовков.

h.request(*method*, *url* [, *body* [, *headers*]])

Отправляет серверу полный запрос HTTP. Аргументы *method* и *url* имеют тот же смысл, что и в методе *h*.putrequest(). В необязательном аргументе *body* передается строка с данными, которые должны быть выгружены после отправки запроса. Если аргумент *body* указан, в заголовке Content-length: автоматически будет указано соответствующее значение. В аргументе *headers* передается словарь, содержащий пары *header*:*value* для передачи методу *h*.putheader().

h.getresponse()

Получает ответ сервера и возвращает экземпляр класса HTTPResponse, с помощью которого можно читать данные. Если экземпляр *h* находится в состоянии, не позволяющем выполнить прием ответа, метод возбуждает исключение ResponseNotReady.

Экземпляр *r* класса HTTPResponse, возвращаемый методом getresponse(), поддерживает следующие методы:

r.read([*size*])

Читает до *size* байтов из ответа сервера. При вызове без аргумента *size* возвращает все данные, полученные в ответ на запрос.

r.getheader(*name* [,*default*])

Получает заголовки ответа. В аргументе *name* передается имя заголовка, а в аргументе *default* — значение по умолчанию, возвращаемое в случае отсутствия заголовка.

r.getheaders()

Возвращает список кортежей (*header*, *value*).

Кроме того, экземпляр *r* класса HTTPResponse обладает следующими атрибутами:

r.version

Версия протокола HTTP, используемая сервером.

r.status

Код состояния HTTP, полученный от сервера.

r.reason

Текст сообщения HTTP, полученный от сервера.

r.length

Количество байтов, оставшихся в ответе.

Исключения

При работе с HTTP-соединениями могут возбуждаться следующие исключения:

Исключение	Описание
HTTPException	Базовый класс всех исключений, имеющих отношение к HTTP.
NotConnected	Ошибка соединения при попытке выполнить запрос.
InvalidURL	Ошибка в адресе URL или в номере порта.
UnknownProtocol	Неизвестный номер протокола HTTP.
UnknownTransferEncoding	Неизвестный формат передачи.
UnimplementedFileMode	Нереализованный режим работы с файлами.
IncompleteRead	Данные приняты не полностью.
BadStatusLine	Принят неизвестный код состояния.

Ниже перечислены исключения, имеющие отношение к состоянию соединений HTTP/1.1. Протокол HTTP/1.1 позволяет передавать несколько запросов/ответов через одно и то же соединение, поэтому существуют дополнительные правила, которые диктуют, когда запросы могут отправляться, а ответы – приниматься. Попытка выполнить операции в недопустимом порядке будет приводить к исключению.

Исключение	Описание
ImproperConnectionState	Базовый класс всех исключений, имеющих отношение к состоянию HTTP-соединений.
CannotSendRequest	Невозможно отправить запрос.
CannotSendHeader	Невозможно отправить заголовки.
ResponseNotReady	Невозможно прочитать ответ.

Пример

Следующий пример демонстрирует применение класса HTTPConnection для выгрузки файла на сервер в виде запроса POST способом, не требующим больших объемов памяти, чего непросто добиться при использовании пакета urllib.

```
import os
try:
    from httplib import HTTPConnection     # Python 2
except ImportError:
    from http.client import HTTPConnection # Python 3

BOUNDARY = "$Python-Essential-Reference$"
CRLF     = '\r\n'

def upload(addr, url, formfields, filefields):
    # Создать разделы для полей формы
    formsections = []
    for name in formfields:
```

```
    section = [
        '--'+BOUNDARY,
        'Content-disposition: form-data; name="%s"' % name,
        '',
        formfields[name]
        ]
    formsections.append(CRLF.join(section)+CRLF)

# Собрать информацию обо всех выгружаемых файлах
fileinfo = [(os.path.getsize(filename), formname, filename)
                    for formname, filename in filefields.items()]

# Создать HTTP-заголовки для каждого файла
filebytes = 0
fileheaders = []
for filesize, formname,filename in fileinfo:
    headers = [
        '--'+BOUNDARY,
        'Content-Disposition: form-data; name="%s"; filename="%s"' % \
                    (formname, filename),
        'Content-length: %d' % filesize,
        ''
        ]
    fileheaders.append(CRLF.join(headers)+CRLF)
    filebytes += filesize

# Закрывающая метка
closing = "--"+BOUNDARY+"--\r\n"

# Определить общую длину запроса
content_size = (sum(len(f) for f in formsections) +
                sum(len(f) for f in fileheaders) +
                filebytes+len(closing))

# Выгрузить его
conn = HTTPConnection(*addr)
conn.putrequest("POST",url)
conn.putheader("Content-type",
               'multipart/form-data; boundary=%s' % BOUNDARY)
conn.putheader("Content-length", str(content_size))
conn.endheaders()

# Отправить все разделы формы
for s in formsections:
    conn.send(s.encode('latin-1'))

# Отправить все файлы
for head,filename in zip(fileheaders,filefields.values()):
    conn.send(head.encode('latin-1'))
    f = open(filename,"rb")
    while True:
        chunk = f.read(16384)
        if not chunk: break
        conn.send(chunk)
    f.close()
conn.send(closing.encode('latin-1'))
```

```
    r = conn.getresponse()
    responsedata = r.read()
    conn.close()
    return responsedata

# Пример: Выгрузить несколько файлов. Удаленный сервер ожидает получить
# поля формы 'name', 'email', 'file_1','file_2' и так далее
# (очевидно, что это будет зависеть от конкретного сервера).
server     = ('localhost', 8080)
url        = '/cgi-bin/upload.py'
formfields = {
    'name'  : 'Dave',
    'email' : 'dave@dabeaz.com'
}
filefields = {
    'file_1' : 'IMG_1008.JPG',
    'file_2' : 'IMG_1757.JPG'
}
resp = upload(server, url,formfields,filefields)
print(resp)
```

Модуль http.server (BaseHTTPServer, CGIHTTPServer, SimpleHTTPServer)

Модуль `http.server` предоставляет различные классы, реализующие функциональность серверов HTTP. В Python 2 содержимое этого модуля разбито на три модуля: `BaseHTTPServer`, `CGIHTTPServer` и `SimpleHTTPServer`.

Класс HTTPServer

Следующий класс реализует простейший сервер HTTP. В Python 2 он объявлен в модуле `BaseHTTPServer`.

`HTTPServer(`*`server_address`*`, `*`request_handler`*`)`

Создает новый объект класса `HTTPServer`. В аргументе *`server_address`* передается кортеж вида (*`host`*, *`port`*), определяющий адрес, на котором сервер будет ожидать поступления запросов на соединение. В аргументе *`request_handler`* передается класс-обработчик, производный от класса `BaseHTTPRequestHandler`, который описывается ниже.

Класс `HTTPServer` является непосредственным наследником класса `TCPServer`, объявленного в модуле `socketserver`. Поэтому, если потребуется реализовать нестандартные операции для работы с сервером HTTP, можно создать свой класс, производный от класса `HTTPServer`, и расширить его. Ниже приводится пример создания многопоточного сервера HTTP, принимающего соединения только из определенной подсети:

```
try:
    from http.server import HTTPServer        # Python 3
    from socketserver import ThreadingMixIn
except ImportError:
    from BaseHTTPServer import HTTPServer     # Python 2
    from SocketServer import ThreadingMixIn
```

```
class MyHTTPServer(ThreadingMixIn,HTTPServer):
    def __init__(self,addr,handler,subnet):
        HTTPServer.__init__(self,addr,handler)
        self.subnet = subnet
    def verify_request(self, request, client_address):
        host, port = client_address
        if not host.startswith(subnet):
            return False
        return HTTPServer.verify_request(self,request,client_address)

# Пример запуска сервера
serv = MyHTTPServer(('',8080), SomeHandler, '192.168.69.')
serv.serve_forever()
```

Класс `HTTPServer` оперирует только низкоуровневым протоколом HTTP. Чтобы сервер действительно мог делать что-то полезное, необходимо указать класс обработчика. Существует два встроенных класса обработчиков и базовый класс, на основе которого можно реализовать собственную обработку. Все они описываются ниже.

Классы SimpleHTTPRequestHandler и CGIHTTPRequestHandler

Имеется возможность использовать два предопределенных класса обработчиков, если необходимо быстро создать простой веб-сервер. Эти классы действуют независимо от любых сторонних веб-серверов, таких как Apache.

`CGIHTTPRequestHandler(`*`request`*`, `*`client_address`*`, `*`server`*`)`

Обслуживает файлы в текущем каталоге и во всех вложенных подкаталогах. Кроме того, обработчик будет выполнять файлы, как сценарии CGI, если они находятся в специальном каталоге **CGI (имена каталогов определяются атрибутом** `cgi_directories` **класса, который по умолчанию содержит список** `['/cgi-bin', '/htbin']`**). Обработчик поддерживает методы** `GET`, `HEAD` **и** `POST`**. Но он не поддерживает переадресацию HTTP (код 302), вследствие чего он может использоваться только для выполнения наиболее простых CGI-приложений. Из соображений безопасности сценарии CGI выполняются с привилегиями пользователя** `nobody`**. В Python 2 определение этого класса находится в модуле** `CGIHTTPServer`**.**

`SimpleHTTPRequestHandler(`*`request`*`, `*`client_address`*`, `*`server`*`)`

Обслуживает файлы в текущем каталоге и во всех вложенных подкаталогах. Этот класс реализует поддержку методов `HEAD` и `GET` запросов. Все исключения `IOError` приводят к выводу сообщения об ошибке `"404 File not found"` **(«404 Файл не найден»). Попытки обратиться к каталогам оканчиваются выводом сообщения об ошибке** `"403 Directory listing not supported"` **(«403 Вывод содержимого каталогов не поддерживается»). В Python 2 определение этого класса находится в модуле** `SimpleHTTPServer`**.**

Оба эти обработчика определяют следующие атрибуты класса, значения которых при желании можно изменять за счет наследования:

`handler``.server_version`

Строка с номером версии, возвращаемая клиенту. По умолчанию имеет строковое значение, такое как `'SimpleHTTP/0.6'`.

handler.extensions_map

Словарь, отображающий расширения файлов в типы MIME. Файлы неопознанного типа относятся к типу 'application/octet-stream'. Ниже приводится пример использования этих классов обработчиков для запуска самостоятельного веб-сервера, поддерживающего возможность выполнения CGI-сценариев:

```
try:
    from http.server import HTTPServer, CGIHTTPRequestHandler # Python 3
except ImportError:
    from BaseHTTPServer import HTTPServer                     # Python 2
    from CGIHTTPServer import CGIHTTPRequestHandler
import os

# Перейти в корневой каталог с документами
os.chdir("/home/httpd/html")

# Запустить сервер CGIHTTP на порту с номером 8080
serv = HTTPServer(("",8080),CGIHTTPRequestHandler)
serv.serve_forever()
```

Класс BaseHTTPRequestHandler

Класс BaseHTTPRequestHandler может использоваться как базовый класс для разработки собственного обработчика, действующего в составе сервера HTTP. Встроенные классы обработчиков, такие как SimpleHTTPRequestHandler и CGIHTTPRequestHandler, также являются производными этого класса. В Python 2 определение этого класса находится в модуле BaseHTTPServer.

BaseHTTPRequestHandler(*request*, *client_address*, *server*)

Базовый класс обработчика, используемый для обработки запросов HTTP. После того как клиент установит соединение и HTTP-заголовки его запроса будут проанализированы, выполняется попытка вызвать метод вида do_*REQUEST*, имя которого конструируется исходя из типа запроса. Например, для обработки запроса типа 'GET' будет вызван метод do_GET(), а для обработки запроса типа 'POST' – метод do_POST(). По умолчанию этот класс ничего не делает, и предполагается, что эти методы должны быть переопределены в подклассах.

Ниже перечислены атрибуты класса BaseHTTPRequestHandler, которые могут быть переопределены в подклассах.

BaseHTTPRequestHandler.server_version

Строка с описанием версии программного обеспечения, которая отправляется клиенту, например 'ServerName/1.2'.

BaseHTTPRequestHandler.sys_version

Версия Python, например 'Python/2.6'.

BaseHTTPRequestHandler.error_message_format

Строка формата, которая используется для сборки сообщений об ошибках, отправляемых клиенту. Строка формата применяется к словарю с атрибутами code, message и explain. Например:

```
'''<head>
   <title>Error response</title>
   </head>
   <body>
   <h1>Ошибка!</h1>
   <p>Код: %(code)d.
   <p>Текст: %(message)s.
   <p>Описание: %(code)s = %(explain)s.
   </body>'''
```

`BaseHTTPRequestHandler.protocol_version`

Версия протокола **HTTP, используемого для передачи ответов. По умолчанию используется значение** `'HTTP/1.0'`.

`BaseHTTPRequestHandler.responses`

Отображение целочисленных кодов ошибок **HTTP в кортежи из двух элементов** (*message, explain*)**, описывающих проблему. Например, код 404 отображается в кортеж** (“Not Found”, “Nothing matches the given URI”)**. Целочисленный код и строки из этого отображения используются при создании сообщений об ошибках, в соответствии со значением атрибута** `error_message_format`**, описанного выше.**

Экземпляр *b* класса `BaseHTTPRequestHandler`, который используется для обработки соединения с клиентом, обладает следующими атрибутами:

Атрибут	Описание
b.client_address	Адрес клиента в виде кортежа (*host, port*).
b.command	Тип запроса, такой как `'GET'`, `'POST'`, `'HEAD'` и другие.
b.path	Путь к объекту запроса, такой как `'/index.html'`.
b.request_version	Строка с номером версии HTTP, использованного для запроса, такая как `'HTTP/1.0'`.
b.headers	Объект отображения с заголовками HTTP. Чтобы проверить присутствие какого-либо заголовка или извлечь его, можно использовать операции над словарями, такие как *headername* `in` *b*.headers или *headerval* = *b*.headers[*headername*].
b.rfile	Поток ввода для получения дополнительных входных данных. Этот атрибут используется, когда клиент выгружает данные на сервер (например, в ходе выполнения запроса `POST`).
b.wfile	Выходной поток, куда записывается ответ, отправляемый клиенту.

Ниже перечислены методы, которые наиболее часто переопределяются в подклассах:

b.`send_error(`*code* `[,` *message*`])`

Отправляет ответ в случае неудачи при обработке запроса. В аргументе *code* передается числовой код ответа HTTP. В аргументе *message* передается дополнительное сообщение об ошибке. Для записи сообщения об ошибке

в журнал вызывается метод log_error(). Этот метод создает законченный ответ с сообщением об ошибке, используя строку формата в атрибуте класса error_message_format, и отправляет ее клиенту. Затем **он закрывает соединение**. После вызова этого метода не должно выполняться никаких других операций.

b.send_response(*code* [, *message*])

Отправляет ответ в случае успешной обработки запроса. Отправляет **строку с ответом HTTP**, за которой следуют заголовки Server и Date. **В аргументе** *code* передается числовой код ответа HTTP, а в аргументе *message* – **дополнительное сообщение**. Для записи запроса в журнал вызывается метод log_request().

b.send_header(*keyword*, *value*)

Выводит заголовок HTTP в выходной поток. В аргументе *keyword* передается имя заголовка, а в аргументе *value* – его значение. Этот метод должен вызываться только после вызова метода send_response().

b.end_headers()

Отправляет пустую строку, как признак окончания последовательности заголовков HTTP.

b.log_request([*code* [, *size*]])

Записывает в журнал успешно обработанный запрос. В аргументе *code* передается код HTTP, а в аргументе *size* – размер ответа в байтах (если доступен). По умолчанию для журналирования вызывает метод log_message().

b.log_error(*format*, ...)

Записывает в журнал сообщение об ошибке. По умолчанию для журналирования вызывает метод log_message().

b.log_message(*format*, ...)

Выводит любые сообщения в поток sys.stderr. В аргументе *format* передается строка формата, которая содержит спецификаторы для любых дополнительных аргументов. В начало каждого сообщения вставляются адрес клиента и текущее время.

Ниже приводится пример сервера **HTTP, который выполняется в отдельном потоке управления и может контролироваться с помощью словаря**, ключи в котором интерпретируются как пути к документам.

```
try:
    from http.server import BaseHTTPRequestHandler, HTTPServer    # Py 3
except ImportError:
    from BaseHTTPServer import BaseHTTPRequestHandler, HTTPServer # Py 2

class DictRequestHandler(BaseHTTPRequestHandler):
    def __init__(self,thedict,*args,**kwargs):
        self.thedict = thedict
        BaseHTTPRequestHandler.__init__(self,*args,**kwargs)

    def do_GET(self):
        key = self.path[1:]       # Отбросить ведущий символ '/'
```

```
            if not key in self.thedict:
                self.send_error(404, "No such key")
            else:
                self.send_response(200)
                self.send_header('content-type','text/plain')
                self.end_headers()
                resp  = "Key : %s\n" % key
                resp += "Value: %s\n" % self.thedict[key]
                self.wfile.write(resp.encode('latin-1'))

# Пример использования
d = {
    'name' : 'Dave',
    'values' : [1,2,3,4,5],
    'email' : 'dave@dabeaz.com'
}
from functools import partial
serv = HTTPServer(("",9000), partial(DictRequestHandler,d))

import threading
d_mon = threading.Thread(target=serv.serve_forever)
d_mon.start()
```

Чтобы опробовать этот пример, запустите сервер и введите в броузере адрес URL, такой как `http://localhost:9000/name` или `http://localhost:9000/values`. Если все действует как надо, вы увидите в окне броузера содержимое словаря.

Кроме всего прочего, этот пример иллюстрирует способ передачи класса обработчика с дополнительными параметрами, которые должны использоваться при создании экземпляра обработчика. Обычно серверы создают обработчики, используя предопределенный набор аргументов, которые передаются методу `__init__()`. Передать дополнительные аргументы можно с помощью функции `functools.partial()`, как показано в этом примере. Эта функция возвращает вызываемый объект, включающий дополнительный параметр, но сохраняет сигнатуру вызова, ожидаемую сервером.

Модуль http.cookies (Cookie)

Модуль `http.cookies` обеспечивает возможность работы с блоками данных cookie на стороне сервера. В Python 2 этот модуль называется `Cookie`.

Блоки данных cookie позволяют управлять состоянием веб-приложений, реализующих поддержку сеансов, аутентификации пользователей и обеспечивающих прочие похожие возможности. Чтобы отправить блок данных cookie броузеру пользователя, сервер HTTP обычно добавляет в ответ заголовок HTTP, как показано ниже:

```
Set-Cookie: session=8273612; expires=Sun, 18-Feb-2001 15:00:00 GMT; \
            path=/; domain=foo.bar.com
```

Альтернативный вариант передачи cookie заключается в том, чтобы внедрить код JavaScript в раздел `<head>` документа HTML:

```
<SCRIPT LANGUAGE="JavaScript">
document.cookie = "session=8273612; expires=Sun, 18-Feb-2001 15:00:00 GMT; \
```

```
    Feb 17; Path=/; Domain=foo.bar.com;"
</SCRIPT>
```

Модуль http.cookies упрощает задачу создания значений cookie, предоставляя специализированный объект, напоминающий словарь, хранящий коллекции значений cookie, известных как *атрибуты* (morsels). Каждый атрибут имеет имя, значение и набор дополнительных атрибутов с метаданными, которые отправляются броузеру {expires, path, comment, domain, max-age, secure, version, httponly}. Имя обычно является простым идентификатором, таким как "name", и не должно совпадать с именами атрибутов метаданных, такими как "expires" или "path". В качестве значений обычно используются короткие строки. Чтобы создать cookie, достаточно просто создать объект, как показано ниже:

```
c = SimpleCookie()
```

Затем в объект можно добавить значения (атрибуты), используя привычную инструкцию присваивания значения элементу словаря:

```
c["session"] = 8273612
c["user"] = "beazley"
```

Метаинформация об атрибутах записывается, как показано ниже:

```
c["session"]["path"] = "/"
c["session"]["domain"] = "foo.bar.com"
c["session"]["expires"] = "18-Feb-2001 15:00:00 GMT"
```

Вывести данные из cookie в виде набора заголовков HTTP можно с помощью метода c.output(). Например:

```
print(c.output())
# Выведет две строки
# Set-Cookie: session=8273612; expires=...; path=/; domain=...
# Set-Cookie: user=beazley
```

Когда броузер отправит cookie обратно серверу HTTP, он будет представлен в виде строки, содержащей пары *key=value*, например: "session=8273612; user=beazley". Дополнительные атрибуты, такие как expires, path и domain, не возвращаются. Строка cookie обычно сохраняется в переменной окружения HTTP_COOKIE, которая доступна в приложениях CGI. Прочитать значения cookie можно следующим способом:

```
c = SimpleCookie(os.environ["HTTP_COOKIE"])
session = c["session"].value
user    = c["user"].value
```

Ниже приводится более подробное описание объектов класса SimpleCookie.

`SimpleCookie([`*input*`])`

Возвращает объект cookie, в котором значения хранятся в виде простых строк.

Экземпляр *c* класса SimpleCookie обладает следующими методами:

c.output([*attrs* [,*header* [,*sep*]]])

Возвращает строку, пригодную для передачи в виде заголовков HTTP. В аргументе *attrs* передается список дополнительных атрибутов, которые должны быть включены в строку ("expires", "path", "domain" и другие). По умолчанию включаются все атрибуты. В аргументе *header* передается имя заголовка HTTP (по умолчанию 'Set-Cookie:'). Аргумент *sep* определяет символ, который должен использоваться для объединения заголовков; по умолчанию содержит символ перевода строки.

c.js_output([*attrs*])

Возвращает строку с программным кодом JavaScript, который устанавливает cookie при выполнении в броузере, поддерживающем JavaScript. В аргументе *attrs* передается список дополнительных атрибутов, которые должны быть включены.

c.load(*rawdata*)

Загружает в объект *c* данные из *rawdata*. Если в аргументе *rawdata* передается строка, предполагается, что она имеет тот же формат, что и переменная окружения HTTP_COOKIE в программах CGI. Если в аргументе *rawdata* передается словарь, каждая пара *key-value* используется для выполнения операции *c*[*key*]=*value*.

Внутри пары *key/value* хранятся в виде экземпляров класса Morsel. Экземпляр *m* класса Morsel своим поведением напоминает словарь и позволяет присваивать значения дополнительным ключам "expires", "path", "comment", "domain", "max-age", "secure", "version" и "httponly". Кроме того, экземпляр *m* атрибута обладает следующими методами и атрибутами:

m.value

Строка, содержащая значение cookie.

m.coded_value

Строка, содержащая кодированное значение cookie, которое посылается или принимается от броузера.

m.key

Имя cookie.

m.set(*key*, *value*, *coded_value*)

Устанавливает значения атрибутов *m*.key, *m*.value и *m*.coded_value, описанных выше.

m.isReservedKey(*k*)

Проверяет, является ли *k* зарезервированным именем, таким как "expires", "path", "domain" и так далее.

m.output([*attrs* [,*header*]])

Возвращает строку заголовка HTTP для данного атрибута *m*. В аргументе *attrs* передается список дополнительных атрибутов, которые должны быть включены ("expires", "path" и другие). В аргументе *header* передается имя заголовка HTTP (по умолчанию 'Set-Cookie:').

m.js_output([*attrs*])

Возвращает строку с программным кодом JavaScript, который устанавливает cookie при выполнении в броузере.

m.OutputString([*attrs*])

Возвращает строку cookie без имен заголовков HTTP или программного кода JavaScript.

Исключения

Если в процессе анализа значений cookie возникнет ошибка, будет возбуждено исключение CookieError.

Модуль http.cookiejar (cookielib)

Модуль http.cookiejar обеспечивает поддержку механизма сохранения и управления данными cookie на стороне клиента. В Python 2 этот модуль называется cookielib.

Основное назначение этого модуля – предоставить объекты, способные хранить cookie так, чтобы их можно было использовать вместе с пакетом urllib, который часто применяется для реализации доступа к документам в Интернете. Например, модуль http.cookiejar может использоваться для сохранения данных cookie и последующей передачи их в запросах. Кроме того, этот модуль может использоваться для работы с файлами, где хранятся данные cookie, например с файлами, которые создаются различными броузерами.

Ниже перечислены классы, объявленные в модуле:

CookieJar()

Объект, который может использоваться для управления значениями cookie, сохранения значений, принятых в результате запросов HTTP, и добавления значений cookie в исходящие запросы HTTP. Значения cookie хранятся в памяти и пропадают, когда экземпляр класса CookieJar утилизируется сборщиком мусора.

FileCookieJar(*filename* [, *delayload*])

Создает экземпляр класса FileCookieJar, который может использоваться для извлечения и сохранения cookie в файле. В аргументе *filename* передается имя файла. Если в аргументе *delayload* передать значение True, будет использоваться прием отложенного доступа к файлу. То есть операции чтения и записи в файл будут выполняться только по требованию.

MozillaCookieJar(*filename* [, *delayload*])

Создает экземпляр класса FileCookieJar, совместимый с файлом cookies.txt, который создает броузер Mozilla.

LWPCookieJar(*filename* [, *delayload*])

Создает экземпляр класса FileCookieJar, совместимый с форматом Set-Cookie3, который используется библиотекой libwww-perl.

На практике достаточно редко приходится работать с атрибутами и методами этих объектов. Если вам потребуется поближе познакомиться с их программным интерфейсом, обращайтесь к электронной документации. Обычно программы просто создают экземпляр одного из описанных классов и подключают его к другому объекту, которому необходима поддержка операций с cookie. Пример такого использования приводится в разделе с описанием модуля `urllib.request` в этой же главе.

Модуль smtplib

Модуль `smtplib` реализует низкоуровневый интерфейс клиента SMTP, который может использоваться для отправки электронной почты по протоколу SMTP, описываемому в RFC 821 и RFC 1869. Этот модуль содержит множество низкоуровневых функций и методов, которые подробно описываются в электронной документации. Тем не менее ниже приводится описание наиболее интересных компонентов этого модуля:

`SMTP([`*`host`* `[, `*`port`*`]])`

Создает объект, представляющий соединение с сервером SMTP. Аргумент *host*, если задан, определяет имя хоста сервера SMTP. В необязательном аргументе *port* передается номер порта. По умолчанию используется порт с номером 25. Если при вызове был указан аргумент *host*, автоматически вызывается метод `connect()`. В противном случае придется вручную вызывать метод `connect()` возвращаемого объекта, чтобы установить соединение.

Экземпляр *s* класса `SMTP` обладает следующими методами:

s`.connect([`*`host`* `[, `*`port`*`]])`

Соединяется с сервером SMTP. Если аргумент *host* отсутствует, соединение устанавливается с локальным компьютером (`'127.0.0.1'`). В необязательном аргументе *port* передается номер порта, который по умолчанию принимается равным 25. Если имя хоста было передано конструктору `SMTP()`, вызывать метод `connect()` не требуется.

s`.login(`*`user`*`, `*`password`*`)`

Регистрируется на сервере, если он требует выполнить аутентификацию. В аргументе *user* передается имя пользователя, а в аргументе *password* – пароль.

s`.quit()`

Закрывает сеанс, отправляя серверу команду `'QUIT'`.

s`.sendmail(`*`fromaddr`*`, `*`toaddrs`*`, `*`message`*`)`

Отправляет электронное письмо серверу. В аргументе *fromaddr* передается строка с адресом отправителя. В аргументе *toaddrs* – список строк с адресами получателей. В аргументе *message* – строка с сообщением в формате, совместимым с требованиями RFC-822. Для создания таких сообщений часто используется пакет `email`. Важно отметить, что несмотря на то, что сообщение передается в виде текстовой строки, она может содержать только допустимые символы ASCII, со значениями в диапазоне от 0 до 127. В про-

тивном случае будет возникать ошибка кодирования. Если вам потребуется отправить сообщение в другой кодировке, такой как UTF-8, сначала преобразуйте его в строку байтов, а затем передайте эту строку в аргументе *message*.

Пример

Ниже демонстрируется, как с помощью модуля `smtplib` можно отправить сообщение:

```
import smtplib
fromaddr = "someone@some.com"
toaddrs = ["recipient@other.com"]
msg = "From: %s\r\nTo: %s\r\n\r\n" % (fromaddr, ",".join(toaddrs))
msg += """
Вкладывайте деньги в акции и виагру!
"""
server = smtplib.SMTP('localhost')
server.sendmail(fromaddr, toaddrs, msg)
server.quit()
```

Пакет urllib

Пакет `urllib` реализует высокоуровневый интерфейс для взаимодействий с серверами HTTP, FTP и доступа к локальным файлам. К типичным приложениям, в которых используется этот пакет, относятся приложения, извлекающие информацию из веб-страниц, автоматизирующие сетевые операции, выполняющие промежуточную обработку, веб-роботы и так далее. Это один из самых динамично развивающихся пакетов, поэтому данное описание не ставит своей целью отображение самых последних новшеств. Вместо этого будут представлены наиболее типичные способы использования пакета.

В Python 2 функциональность пакета `urllib` находится в различных модулях, таких как `urllib`, `urllib2`, `urlparse` и `robotparser`. В Python 3 все функциональные возможности были реорганизованы и сосредоточены в пакете `urllib`.

Модуль urllib.request (urllib2)

Модуль `urllib.request` реализует функции и классы, позволяющие открывать и извлекать данные по заданному адресу URL. В Python 2 эта функциональность находится в модуле `urllib2`.

Чаще всего этот модуль используется для извлечения данных из веб-серверов с помощью протокола HTTP. Например, следующий фрагмент демонстрирует простейший способ получения веб-страницы:

```
try:
    from urllib.request import urlopen # Python 3
except ImportError:
    from urllib2 import urlopen        # Python 2
```

```
u = urlopen("http://docs.python.org/3.0/library/urllib.request.html")
data = u.read()
```

Безусловно, при взаимодействии с настоящими серверами возникают самые разные сложности. Например, может потребоваться побеспокоиться о прокси-сервере, об аутентификации, о cookie, о предоставлении информации о версии клиентского программного обеспечения и о многом другом. Все эти возможности поддерживаются пакетом, хотя это и усложняет программный код (читайте далее).

Функция urlopen() и запросы

Самый простой способ выполнить запрос – вызвать функцию `urlopen()`.

`urlopen(`*`url`* `[,` *`data`* `[,` *`timeout`*`]])`

Открывает адрес URL *`url`* и возвращает объект, напоминающий файл, который может использоваться для чтения полученных данных. В аргументе *`url`* можно передать строку с адресом URL или экземпляр класса `Request`, который описывается ниже. В аргументе *`data`* передается строка в формате URL с данными формы, которые должны быть выгружены на сервер. Если функция получает этот аргумент, вместо HTTP-метода `'GET'` (по умолчанию) она будет использовать метод `'POST'`. Строка с данными обычно создается с помощью такой функции, как `urllib.parse.urlencode()`. В аргументе *`timeout`* передается предельное время ожидания для всех блокирующих операций, используемых внутренней реализацией.

Объект *`u`*, напоминающий файл, возвращаемый функцией `urlopen()`, поддерживает следующие методы:

Метод	Описание
`u.read([nbytes])`	Читает *`nbytes`* байтов данных и возвращает их в виде строки байтов.
`u.readline()`	Читает одну строку текста и возвращает в виде строки байтов.
`u.readlines()`	Читает все строки и возвращает список.
`u.fileno()`	Возвращает целочисленный дескриптор файла.
`u.close()`	Закрывает соединение.
`u.info()`	Возвращает объект отображения с метаинформацией об адресе URL. Для HTTP сюда входят заголовки HTTP, полученные от сервера. Для FTP добавляется заголовок `'content-length'`. Для локальных файлов добавляется информация о дате и заголовки `'content-length'` и `'content-type'`.
`u.getcode()`	Возвращает код ответа HTTP в виде целого числа. Например, в случае успеха возвращает число `200`, если файл не найден – число `404`.
`u.geturl()`	Возвращает действительный адрес URL полученных данных, при этом учитываются все возникающие переадресации.

Важно подчеркнуть, что объект *u*, напоминающий файл, действует в двоичном режиме. Если данные ответа должны обрабатываться как текст, их необходимо декодировать с применением модуля codecs или какого-либо другого инструмента.

Если в процессе загрузки данных возникнет ошибка, будет возбуждено исключение URLError. Сюда входят ошибки, связанные с протоколом HTTP, такие как «доступ запрещен» или «требуется аутентификация». Вместе с кодами подобных ошибок серверы обычно возвращают дополнительную информацию. Для ее получения можно использовать объект исключения – как файл, доступный для чтения. Например:

```
try:
    u = urlopen("http://www.python.org/perl.html")
    resp = u.read()
except HTTPError as e:
    resp = e.read()
```

Наиболее типичной ошибкой, возникающей при использовании функции urlopen(), является попытка обращения к веб-страницам через прокси-сервер. Например, на предприятии весь трафик может направляться через прокси-сервер, в этом случае прямые запросы будут терпеть неудачу. Если прокси-сервер не требует выполнять процедуру аутентификации, эту ошибку можно исправить, просто добавив переменную окружения HTTP_PROXY в словарь os.environ. Например, os.environ['HTTP_PROXY'] = 'http://example.com:12345'.

В самом простом случае в аргументе *url* функции urlopen() передается строка, такая как 'http://www.python.org'. Если необходимо реализовать более сложную операцию, например изменить заголовки HTTP-запроса, можно создать экземпляр класса Request и использовать его в качестве значения аргумента *url*.

Request(*url* [, *data* [, *headers* [, *origin_req_host* [, *unverifiable*]]]])

Создает экземпляр класса Request. Аргумент *url* определяет адрес URL (например, 'http://www.foo.bar/spam.html'). В аргументе *data* передаются данные в формате URL, предназначенные для передачи на сервер вместе с HTTP-запросом. Если этот аргумент присутствует в вызове конструктора, тип HTTP-запроса изменяется с 'GET' на 'POST'. В аргументе *headers* передается словарь, содержащий пары *key-value*, представляющие содержимое HTTP-заголовков. В аргументе *origin_req_host* обычно передается имя хоста, откуда выполняется запрос. В аргументе *unverifiable* передается значение True, если выполняется запрос для адреса URL, не поддающегося проверке. Адрес URL, не поддающийся проверке, неформально определяется, как адрес URL, который не был введен непосредственно пользователем, – например, адрес URL внутри страницы, которая загружает изображение. По умолчанию в аргументе *unverifiable* передается False.

Экземпляр *r* класса Request обладает следующими методами:

r.add_data(*data*)

Добавляет данные в запрос. Если запрос является HTTP-запросом, его тип изменяется на 'POST'. В аргументе *data* передаются данные в формате URL,

как рассказывалось в описании функции `Request()`. Этот метод не добавляет новые данные к старым, а просто замещает старые данные значением аргумента *data*.

r.`add_header(key, val)`

Добавляет заголовок в запрос. В аргументе *key* передается имя заголовка, а в аргументе *val* – значение. Оба значения должны быть строками.

r.`add_unredirected_header(`*key, val*`)`

Добавляет заголовок, который будет исключен из запроса при переадресации. Аргументы *key* и *val* имеют тот же смысл, что и в методе `add_header()`.

r.`get_data()`

Возвращает данные запроса (если имеются).

r.`get_full_url()`

Возвращает полный адрес URL запроса.

r.`get_host()`

Возвращает имя хоста, которому будет отправлен запрос.

r.`get_method()`

Возвращает метод HTTP, который может быть `'GET'` или `'POST'`.

r.`get_origin_req_host()`

Возвращает имя хоста, откуда выполняется запрос.

r.`get_selector()`

Возвращает часть URL, представляющую путь (например, `'/index.html'`).

r.`get_type()`

Возвращает тип URL (например, `'http'`).

r.`has_data()`

Возвращает `True`, если запрос содержит дополнительные данные.

r.`is_unverifiable()`

Возвращает `True`, если запрос не может быть проверен.

r.`has_header(`*header*`)`

Возвращает `True`, если запрос содержит заголовок *header*.

r.`set_proxy(`*host, type*`)`

Подготавливает запрос для передачи через прокси-сервер. Замещает оригинальное имя хоста в запросе значением *host*, а оригинальный тип запроса – типом *type*. Часть URL, представляющая путь, остается неизменной.

Ниже приводится пример, где с помощью объекта класса `Request` изменяется заголовок `'User-Agent'`, используемый функцией `urlopen()`. Этот прием можно использовать, чтобы заставить сервер думать, что соединение устанавливается с помощью броузера Internet Explorer, Firefox или какого-нибудь другого.

```
headers = {
  'User-Agent':
      'Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1; .NET CLR 2.0.50727)'
}

r = Request("http://somedomain.com/",headers=headers)
u = urlopen(r)
```

Создание собственных объектов доступа к ресурсам

Базовая функция `urlopen()` не обеспечивает поддержку аутентификации, cookie и других дополнительных особенностей протокола HTTP. Чтобы добавить эту поддержку, необходимо с помощью функции `build_opener()` создать свой собственный объект доступа к ресурсам:

`build_opener([`*handler1* `[,` *handler2*`, ... ]])`

Конструирует нестандартный объект доступа к адресам URL. Аргументы *handler1*, *handler2* и так далее являются экземплярами специальных классов обработчиков. Назначение этих обработчиков состоит в том, чтобы добавить различные возможности в возвращаемый объект. Ниже перечислены все доступные классы обработчиков:

Обработчик	Описание
`CacheFTPHandler`	Обработчик доступа к ресурсам FTP с постоянными соединениями
`FileHandler`	Открывает локальные файлы
`FTPHandler`	Открывает URL через FTP
`HTTPBasicAuthHandler`	Простой обработчик аутентификации HTTP
`HTTPCookieProcessor`	Реализует обработку cookie
`HTTPDefaultErrorHandler`	Обрабатывает ошибки HTTP, возбуждая исключение `HTTPError`
`HTTPDigestAuthHandler`	Реализует дайджест-аутентификацию HTTP
`HTTPHandler`	Открывает URL через HTTP
`HTTPRedirectHandler`	Обрабатывает переадресацию HTTP
`HTTPSHandler`	Открывает URL через безопасный HTTP
`ProxyHandler`	Переадресует запросы через прокси-сервер
`ProxyBasicAuthHandler`	Реализует простую аутентификацию на прокси-сервере
`ProxyDigestAuthHandler`	Реализует дайджест-аутентификацию на прокси-сервере
`UnknownHandler`	Обработчик неизвестных адресов URL

По умолчанию объект доступа к ресурсам всегда создается с обработчиками `ProxyHandler`, `UnknownHandler`, `HTTPHandler`, `HTTPSHandler`, `HTTPDefaultErrorHandler`, `HTTPRedirectHandler`, `FTPHandler`, `FileHandler` и `HTTPErrorProcessor`. Эти обработчики образуют базовый уровень функциональности. Дополнитель-

ные обработчики, передаваемые в аргументах, добавляются в этот список. Однако, если какой-либо из дополнительных обработчиков имеет тот же тип, что и обработчик по умолчанию, он получает преимущество. Например, если добавить экземпляр класса HTTPHandler или его производных, он будет использоваться вместо обработчика по умолчанию.

Объект, возвращаемый функцией build_opener(), обладает методом open(*url* [, *data* [, *timeout*]]), который применяется для открытия адреса URL, в соответствии с правилами, определяемыми всеми обработчиками. Метод open() принимает те же аргументы, что и функция urlopen().

install_opener(*opener*)

Устанавливает объект *opener* как глобальный инструмент открытия адресов URL, используемый функцией urlopen(). В аргументе *opener* обычно передается объект, созданный функцией build_opener().

В следующих нескольких разделах рассказывается, как создавать свои объекты доступа к ресурсам для некоторых наиболее типичных случаев, которые могут возникнуть при использовании модуля urlib.request.

Аутентификация паролем

Для работы с запросами, требующими аутентификации паролем, можно создать свой объект доступа к ресурсу, объединяющий в той или иной комбинации обработчики HTTPBasicAuthHandler, HTTPDigestAuthHandler, ProxyBasicAuthHandler или ProxyDigestAuthHandler. Каждый из этих обработчиков обладает следующим методом, позволяющим указать пароль:

h.add_password(*realm*, *uri*, *user*, *passwd*)

Добавляет имя пользователя и пароль для заданной области *realm* и URI *uri*. Во всех аргументах передаются строковые значения. Аргумент *uri* может быть последовательностью идентификаторов ресурсов URI, в этом случае имя пользователя и пароль применяются ко всем URI в последовательности. В аргументе *realm* передается имя или описание области, ассоциированной с аутентификацией. Значение этого аргумента зависит от удаленного сервера. Как правило, для группы взаимосвязанных веб-страниц выбирается общее имя области. В аргументе *uri* передается базовая часть адреса URL, для доступа к которому требуется аутентификация. Типичными значениями аргументов *realm* и *uri* могли бы быть ('Administrator', 'http://www.somesite.com'). Аргументы *user* и *password* определяют имя пользователя и пароль соответственно.

Ниже приводится пример создания объекта доступа к ресурсу с поддержкой простой аутентификации:

```
auth = HTTPBasicAuthHandler()
auth.add_password("Administrator",
                  "http://www.secretlair.com","drevil","12345")

# Создать объект доступа с поддержкой аутентификации
opener = build_opener(auth)

# Открыть URL
u = opener.open("http://www.secretlair.com/evilplan.html")
```

HTTP cookie

Для работы с cookie можно создать объект доступа к ресурсу с включенным в него обработчиком HTTPCookieProcessor. Например:

```
cookiehand = HTTPCookieProcessor()
opener = build_opener(cookiehand)
u = opener.open("http://www.example.com/")
```

По умолчанию экземпляр класса HTTPCookieProcessor использует объект класса CookieJar, определение которого находится в модуле http.cookiejar. Передавая конструктору HTTPCookieProcessor различные типы объектов CookieJar, можно организовать самые разные виды обработки cookie. Например:

```
cookiehand = HTTPCookieProcessor(
                http.cookiejar.MozillaCookieJar("cookies.txt")
             )
opener = build_opener(cookiehand)
u = opener.open("http://www.example.com/")
```

Прокси-серверы

Если запрос требуется направлять через прокси-сервер, можно воспользоваться экземпляром класса ProxyHandler.

ProxyHandler([*proxies*])

Создает обработчик, который переадресует запросы прокси-серверу. В аргументе *proxies* передается словарь, отображающий названия протоколов (например, 'http', 'ftp' и так далее) в адреса URL соответствующих прокси-серверов.

Следующий пример демонстрирует, как задействовать этот обработчик:

```
proxy = ProxyHandler({'http': 'http://someproxy.com:8080/'}
auth = HTTPBasicAuthHandler()
auth.add_password("realm","host", "username", "password")
opener = build_opener(proxy, auth)

u = opener.open("http://www.example.com/doc.html")
```

Модуль urllib.response

Это внутренний модуль, который реализует объекты, напоминающие файлы, возвращаемые функциями из модуля urllib.request. Не имеет общедоступного программного интерфейса.

Модуль urllib.parse

Модуль urllib.parse используется для работы со строками URL, такими как "http://www.python.org".

Анализ адресов URL (модуль urlparse в Python 2)

В общем случае адрес URL имеет вид: "*scheme*://*netloc*/*path*;*parameters*?*query*#*fragment*". Кроме того, часть *netloc* может включать номер порта

"*hostname*:*port*" или информацию для аутентификации пользователя в виде "*user*:*pass*@*hostname*". Следующая функция используется для анализа адресов URL:

`urlparse(`*`urlstring`* `[,` *`default_scheme`* `[,` *`allow_fragments`*`]])`

Анализирует адрес URL в строке *urlstring* и возвращает объект класса `ParseResult`. Аргумент *default_scheme* определяет протокол ("`http`", "`ftp`" и так далее) по умолчанию, на тот случай, если он явно не указан в URL. Если в аргументе *allow_fragments* передать ноль, поиск идентификаторов фрагментов выполняться не будет.

Экземпляр *r* класса `ParseResult` — это именованный кортеж вида (*scheme*, *netloc*, *path*, *parameters*, *query*, *fragment*), но дополнительно он обладает следующими атрибутами:

Атрибут	Описание
r.scheme	Спецификатор протокола (например, '`http`')
r.netloc	Спецификатор местоположения в сети (например, '`www.python.org`')
r.path	Путь (например, '`/index.html`')
r.params	Параметры для последнего элемента в пути
r.query	Строка запроса (например, '`name=Dave&id=42`')
r.fragment	Идентификатор фрагмента без ведущего символа '`#`'
r.username	Имя пользователя, если спецификатор местоположения в сети имеет вид '*username*:*password*@*hostname*'
r.password	Пароль из спецификатора местоположения в сети
r.hostname	Имя хоста из спецификатора местоположения в сети
r.port	Номер порта, если спецификатор местоположения в сети имеет вид '*hostname*:*port*'

Объект класса `ParseResult` можно преобразовать обратно в строку URL, вызвав метод *r*.`geturl()`.

`urlunparse(`*`parts`*`)`

Конструирует строку URL из представления в виде кортежа, полученного вызовом функции `urlparse()`. В аргументе *parts* передается кортеж или итерируемый объект с шестью элементами.

`urlsplit(`*`url`* `[,` *`default_scheme`* `[,` *`allow_fragments`*`]])`

То же, что и функция `urlparse()`, за исключением того, что часть URL *parameters* остается без изменений в составе пути. Это позволяет анализировать строки URL, где параметры могут присоединяться к отдельным компонентам пути, например: '*scheme*://*netloc*/*path1*;*param1*/*path2*;*param2*/*path3*?*query*#*fragment*'. Возвращает экземпляр класса `SplitResult`, который является именованным кортежем вида (*scheme*, *netloc*, *path*, *query*, *fragment*). Кроме того, экземпляр *r* класса `SplitResult` обладает следующими атрибутами:

Атрибут	Описание
r.scheme	Спецификатор протокола (например, 'http')
r.netloc	Спецификатор местоположения в сети (например, 'www.python.org')
r.path	Путь (например, '/index.html')
r.query	Строка запроса (например, 'name=Dave&id=42')
r.fragment	Идентификатор фрагмента без ведущего символа '#'
r.username	Имя пользователя, если спецификатор местоположения в сети имеет вид '*username*:*password*@*hostname*'
r.password	Пароль из спецификатора местоположения в сети
r.hostname	Имя хоста из спецификатора местоположения в сети
r.port	Номер порта, если спецификатор местоположения в сети имеет вид '*hostname*:*port*'

Экземпляр класса SplitResult можно преобразовать обратно в строку URL, вызвав метод *r*.geturl().

urlunsplit(parts)

Конструирует строку URL из представления в виде кортежа, полученного вызовом функции urlsplit(). В аргументе *parts* передается кортеж или итерируемый объект с пятью элементами адреса URL.

urldefrag(*url*)

Возвращает кортеж (*newurl*, *fragment*), где поле *newurl* содержит часть аргумента *url* без спецификатора фрагмента, а поле *fragment* содержит спецификатор фрагмента (если имеется). Если в строке *url* отсутствует спецификатор фрагмента, то в поле *newurl* возвращается значение аргумента *url*, а в поле *fragment* – пустая строка.

urljoin(*base*, *url* [, *allow_fragments*])

Конструирует абсолютный адрес URL, объединяя базовый адрес URL *base* и относительный *url*. Аргумент *allow_fragments* имеет тот же смысл, что и в функции urlparse(). Если последний компонент базового адреса URL не является каталогом, он удаляется.

parse_qs(*qs* [, *keep_blank_values* [, *strict_parsing*]])

Анализирует строку запроса *qs* (в формате application/x-www-form-urlencoded) и возвращает словарь, ключами которого являются имена переменных запроса, а значениями – списки значений, определенных для каждого имени. В аргументе *keep_blank_values* передается логический флаг, управляющий обработкой пустых значений. Если в этом аргументе передать True, пустые значения будут включаться в словарь в виде пустых строк. Если передать False (по умолчанию), они будут отбрасываться. В аргументе *strict_parsing* передается логический флаг. Если этот флаг имеет значение True, все ошибки, возникающие при анализе, будут преобразовываться в исключение ValueError. По умолчанию ошибки просто игнорируются.

`parse_qsl(qs [, keep_blank_values [, strict_parsing]])`

То же, что и `parse_qs()`, за исключением того, что возвращает список кортежей (*name*, *value*), где поле *name* представляет имя переменной строки запроса, а поле *value* — значение.

Кодирование адресов URL (модуль urllib в Python 2)

Ниже перечислены функции, которые используются для кодирования/декодирования данных, составляющих строки URL.

`quote(string [, safe [, encoding [, errors]]])`

Замещает специальные символы в строке *string* экранированными последовательностями, пригодными для включения в строку URL. Алфавитные символы, цифры, символ подчеркивания (_), запятая (,), точка (.) и дефис (-) никогда не изменяются. Все остальные символы преобразуются в экранированные последовательности вида '*%xx*'. В аргументе *safe* можно передать строку с дополнительными символами, которые не должны экранироваться; аргумент по умолчанию имеет значение '/'. Аргумент *encoding* определяет кодировку символов, не входящих в диапазон символов ASCII. По умолчанию используется кодировка 'utf-8'. Аргумент *errors* определяет политику обработки ошибок кодирования и по умолчанию имеет значение 'strict'. Аргументы *encoding* и *errors* **могут использоваться только в Python 3.**

`quote_plus(string [, safe [, encoding [, errors]]])`

Вызывает функцию `quote()` **и дополнительно замещает все пробелы знаками** плюс (+). Аргументы *string* и *safe* имеют тот же смысл, что и в функции `quote()`. Аргументы *encoding* и *errors* имеют тот же смысл, что и в функции `quote()`.

`quote_from_bytes(bytes [, safe])`

То же, что и `quote()`, но принимает строку байтов и не выполняет кодирования символов. Возвращает текстовую строку. Доступна только в Python 3.

`unquote(string [, encoding [, errors]])`

Замещает экранированные последовательности вида '*%xx*' их односимвольными эквивалентами. Аргументы *encoding* и *errors* определяют кодировку символов и политику обработки ошибок декодирования данных в экранированных последовательностях '*%xx*'. По умолчанию используется кодировка 'utf-8' и политика обработки ошибок 'replace'. Аргументы *encoding* и *errors* могут использоваться только в Python 3.

`unquote_plus(string [, encoding [, errors]])`

Действует подобно функции `unquote()`, но дополнительно замещает знаки плюс (+) пробелами.

`unquote_to_bytes(string)`

То же, что и `unquote()`, но не выполняет кодирование символов и возвращает строку байтов.

`urlencode(query [, doseq])`

Преобразует параметры запроса *query* в строку, пригодную для добавления в строку URL или для выгрузки на сервер, в составе запроса `POST`. В аргументе *query* допускается передавать словарь или последовательность кортежей (*key, value*). Возвращаемая строка состоит из последовательности пар '*key=value*', разделенных символом '&', где оба значения, *key* и *value*, обрабатываются с помощью функции `quote_plus()`. В аргументе *doseq* передается логический флаг, который должен иметь значение `True`, если какое-либо из значений в *query* является последовательностью из нескольких значений для одного и того же ключа *key*. В этом случае для каждого значения *v* будет создана отдельная подстрока '*key=v*'.

Примеры

Следующие примеры демонстрируют, как преобразовать словарь с параметрами запроса в строку URL, пригодную для выполнения HTTP-запроса `GET`, и как выполнить анализ строки URL:

```
try:
    from urllib.parse import urlparse, urlencode, parse_qsl # Python 3
except ImportError:
    from urlparse import urlparse, parse_qsl                # Python 2
    from urllib import urlencode

# Пример создания строки URL из словаря с переменными запроса
form_fields = {
    'name'  : 'Dave',
    'email' : 'dave@dabeaz.com',
    'uid'   : '12345'
}
form_data = urlencode(form_fields)
url = "http://www.somehost.com/cgi-bin/view.py?"+form_data

# Пример разложения строки URL на компоненты
r = urlparse(url)
print(r.scheme)      # 'http'
print(r.netloc)      # 'www.somehost.com'
print(r.path)        # '/cgi-bin/view.py'
print(r.params)      # ''
print(r.query)       # 'uid=12345&name=Dave&email=dave%40dabeaz.com'
print(r.fragment)    # ''

# Извлечь данные запроса
parsed_fields = dict(parse_qsl(r.query))
assert form_fields == parsed_fields
```

Модуль urllib.error

Модуль `urllib.error` содержит определения исключений, используемых пакетом `urllib`.

`ContentTooShort`

Возбуждается, когда объем загруженных данных оказывается меньше, чем ожидалось (значение в заголовке `'Content-Length'`). В Python 2 определяется в модуле `urllib`.

`HTTPError`

Возбуждается в случае появления проблем, имеющих отношение к протоколу HTTP. Подобные ошибки могут использоваться как средства оповещения, например, о необходимости аутентификации. Данное исключение может также использоваться как объект файла, для чтения данных, ассоциированных с ошибкой и возвращаемых сервером. Является производным от класса `URLError`. В Python 2 определяется в модуле `urllib2`.

`URLError`

Возбуждается обработчиками при обнаружении проблем. Является производным от класса `IOError`. Дополнительную информацию о проблеме можно получить из атрибута `reason` экземпляра исключения. В Python 2 определяется в модуле `urllib2`.

Модуль urllib.robotparser (robotparser)

Модуль `urllib.robotparser` (в Python 2 называется `robotparser`) используется для получения и анализа содержимого файлов `'robots.txt'`, используемых для управления действиями веб-роботов. Дополнительную информацию об использовании можно найти в электронной документации.

Примечания

- Опытные пользователи могут настраивать поведение пакета `urllib` в весьма широких пределах, включая создание новых типов объектов доступа к ресурсам, обработчиков, запросов, протоколов и так далее. Эта тема выходит далеко за рамки данного раздела, однако все необходимые подробности можно найти в электронной документации.
- Пользователи Python 2 должны заметить, что функция `urllib.urlopen()`, получившая широкое применение, официально была не рекомендована к использованию в Python 2.6 и исключена в Python 3. Вместо функции `urllib.urlopen()` следует использовать функцию `urllib2.urlopen()`, которая обеспечивает ту же функциональность, что и функция `urllib.request.urlopen()`, описанная здесь.

Пакет xmlrpc

Пакет `xmlrpc` содержит модули, предназначенные для реализации серверов и клиентов, действующих по протоколу XML-RPC. Протокол *XML-RPC* – это механизм вызова удаленных процедур, использующий формат XML для представления данных и протокол HTTP – в качестве транспортного. Сам протокол XML-RPC не является чем-то особенным, характерным толь-

ко для программ на языке Python, поэтому не исключается возможность с помощью этих модулей организовать взаимодействие с программами, написанными на других языках программирования. Дополнительную информацию о протоколе XML-RPC можно получить на сайте проекта *http://www.xmlrpc.com*.

Модуль xmlrpc.client (xmlrpclib)

Модуль `xmlrpc.client` используется для разработки клиентов XML-RPC. В Python 2 этот модуль называется `xmlrpclib`. **Чтобы программа могла выступать в роли клиента, она должна создать экземпляр класса** `ServerProxy`:

```
ServerProxy(uri [, transport [, encoding [, verbose [, allow_none
[, use_datetime]]]])
```

В аргументе *uri* передается адрес удаленного сервера XML-RPC, например "`http://www.foo.com/RPC2`". В случае необходимости в строку URI допускается добавлять информацию для аутентификации в виде: "`http://`*user*`:`*pass*`@`*host*`:`*port*`/`*path*", где подстрока *user*`:`*pass* содержит имя пользователя и пароль. Данная информация кодируется в формате base-64 и помещается в заголовок `'Authorization:'` транспортного протокола. Если **интерпретатор** Python был собран с поддержкой OpenSSL, в качестве транспортного протокола можно использовать HTTPS. Аргумент *transport* определяет фабричную функцию создания внутреннего транспортного объекта, используемого для организации низкоуровневых взаимодействий. Этот аргумент используется, только если в качестве транспортного протокола используется другой протокол, отличный от HTTP или HTTPS. В обычных ситуациях этот аргумент практически никогда не требуется (за дополнительными подробностями обращайтесь к электронной документации). Аргумент *encoding* **определяет кодировку символов. По умолчанию используется** кодировка UTF-8. Если в аргументе *verbose* передать значение `True`, в процессе работы объекта будет выводиться дополнительная отладочная информация. Если в аргументе `allow_none` передать значение `True`, удаленному серверу можно будет отправлять значение `None`. По умолчанию такая возможность запрещена, потому что она поддерживается не во всех языках программирования. В аргументе `use_datetime` передается логический флаг. Если он имеет значение `True`, для представления даты и времени будет использоваться модуль `datetime`. По умолчанию принимает значение `False`.

Экземпляр *s* класса `ServerProxy` обеспечивает доступ ко всем методам удаленного сервера. Эти методы доступны как атрибуты объекта *s*. Например, ниже демонстрируется получение текущего времени с удаленного сервера, предоставляющего такую возможность:

```
>>> s = ServerProxy("http://www.xmlrpc.com/RPC2")
>>> s.currentTime.getCurrentTime()
<DateTime u'20051102T20:08:24' at 2c77d8>
>>>
```

В значительной степени вызовы RPC выполняются так же, как вызовы обычных функций на языке Python. **Однако протокол XML-RPC поддер-**

живает ограниченное количество типов аргументов и возвращаемых значений:

Тип XML-RPC	Эквивалент в языке Python
логический	`True` и `False`
целое число	`int`
число с плавающей точкой	`float`
строка	Обычные строки или строки Юникода (могут содержать только символы, допустимые для использования в документах XML)
массив	Любые последовательности, содержащие значения типов, допустимых в XML-RPC
структура	Словарь со строковыми ключами и значениями допустимых типов
дата	Дата и время (`xmlrpc.client.DateTime`)
двоичный	Двоичные данные (`xmlrpc.client.Binary`)

Когда в результате вызова удаленной процедуры возвращается значение даты, она сохраняется в виде экземпляра *d* класса `xmlrpc.client.DateTime`. Атрибут *d*.`value` содержит строку с датой в формате **ISO 8601. Чтобы превратить ее в кортеж, совместимый с модулем** `time`, можно воспользоваться методом *d*.`timetuple()`. Когда возвращаются двоичные данные, они сохраняются в экземпляре *b* класса `xmlrpc.client.Binary`. Атрибут *b*.`data` содержит строку байтов с данными. Под строками подразумеваются строки, состоящие из символов Юникода, и вам придется самостоятельно побеспокоиться о соответствующей кодировке. Передача сырых строк байтов в Python 2 возможна при условии, что они будут содержать только символы ASCII. Чтобы ликвидировать этот недостаток, эти строки можно предварительно преобразовать в строки Юникода.

Если в вызове **RPC использовать аргументы недопустимых типов, это может** привести к появлению исключения `TypeError` или `xmlrpclib.Fault`.

Если удаленный сервер **XML-RPC поддерживает возможность интроспекции**, появляется возможность использовать следующие методы:

s.`system.listMethods()`

Возвращает список строк с именами всех методов, предоставляемых сервером XML-RPC.

s.`methodSignatures(`*name*`)`

Возвращает список всех сигнатур вызова метода с именем *name*. Каждая сигнатура представлена списком типов в виде строк, разделенных запятыми (например, `'string, int, int'`), где первый элемент соответствует типу возвращаемого значения, а остальные – типам аргументов. В случае поддержки сервером механизма перегрузки методов может возвращаться не-

сколько сигнатур. Серверы XML-RPC, реализованные на языке Python, обычно возвращают пустые сигнатуры, что обусловлено динамической типизацией функций и методов.

s.methodHelp(*name*)

Возвращает строку документирования для метода *name*, описывающую порядок его использования. Строка документирования может содержать разметку HTML. Если описание недоступно, возвращается пустая строка.

Ниже перечислены вспомогательные функции, предоставляемые модулем xmlrpclib:

boolean(*value*)

На основе значения *value* создает логический объект XML-RPC. Эта функция существовала еще до того, как в языке Python появился логический тип, поэтому данную функцию часто можно встретить в старых программах.

Binary(*data*)

Создает объект XML-RPC с двоичными данными. В аргументе *data* передается строка с двоичными данными. Возвращает экземпляр класса Binary. Возвращаемый объект автоматически кодируется/декодируется в формат base-64 в процессе передачи. Извлечь двоичные данные из экземпляра *b* класса Binary можно с помощью атрибута *b*.data.

DateTime(*daytime*)

Создает объект XML-RPC, содержащий дату. В аргументе *daytime* можно передать либо строку с датой в формате ISO 8601, либо кортеж, возвращаемый функцией time.localtime(), либо экземпляр типа datetime, поддерживаемого модулем datetime.

dumps(*params* [, *methodname* [, *methodresponse* [, *encoding* [, *allow_none*]]]])

Преобразует параметры *params* в запрос или ответ XML-RPC. В аргументе *params* передается кортеж с аргументами аргументов или экземпляр исключения Fault. В аргументе *methodname* передается имя метода в виде строки. В аргументе *methodresponse* передается логический флаг. Если этот флаг имеет значение True, результатом функции будет ответ XML-RPC. В этом случае в аргументе *params* должно передаваться только одно значение. Аргумент *encoding* определяет кодировку текста в генерируемом фрагменте XML. По умолчанию используется кодировка UTF-8. В аргументе *allow_none* передается флаг, который определяет, поддерживается ли значение None как тип параметра. Значение None явно не упоминается в спецификации протокола XML-RPC, но многие серверы поддерживают его. По умолчанию аргумент *allow_none* принимает значение False.

loads(*data*)

Преобразует данные *data*, содержащиеся в запросе или в ответе XML-RPC, в кортеж (*params*, *methodname*), где поле *params* содержит кортеж параметров, а поле *methodname* – строку с именем метода. Если запрос содержит не фактическое значение, а представление ошибки, возбуждается исключение Fault.

`MultiCall(`*`server`*`)`

Создает объект класса `MultiCall`, позволяющий объединить несколько запросов XML-RPC и отправить их как единый запрос. Этот прием может способствовать повышению производительности, когда одному и тому же серверу RPC требуется отправить множество различных запросов. В аргументе *server* передается экземпляр класса `ServerProxy`, представляющего соединение с удаленным сервером. Возвращаемый объект класса `MultiCall` может использоваться точно так же, как и объект класса `ServerProxy`. Однако, вместо немедленного выполнения, вызовы удаленных методов ставятся в очередь, пока объект класса `MultiCall` не будет вызван как функция. Как только это произойдет, он отправит запрос RPC. В результате выполнения эта операция вернет генератор, который последовательно будет возвращать результат каждой операции RPC. Обратите внимание, что функция `MultiCall()` может вызываться, только если удаленный сервер поддерживает метод `system.multicall()`.

Ниже приводится пример, иллюстрирующий использование объекта `MultiCall`:

```
multi = MultiCall(server)
multi.foo(4,6,7)                # Удаленный метод foo
multi.bar("hello world")        # Удаленный метод bar
multi.spam()                    # Удаленный метод spam
# Теперь выполнить фактический запрос XML-RPC и получить результаты
foo_result, bar_result, spam_result = multi()
```

Исключения

В модуле `xmlrpc.client` определены следующие исключения:

`Fault`

Свидетельствует об ошибке XML-RPC. Атрибут `faultCode` содержит строку с типом ошибки. Атрибут `faultString` содержит текстовое описание ошибки.

`ProtocolError`

Свидетельствует о проблемах, связанных с сетью, например об ошибке в адресе URL или о проблемах с соединением. Атрибут `url` содержит URI, где произошла ошибка. Атрибут `errcode` содержит код ошибки. Атрибут `errmsg` содержит строку с текстовым описанием ошибки. Атрибут `headers` содержит HTTP-заголовки запроса, вызвавшего ошибку.

Модуль xmlrpc.server (SimpleXMLRPCServer, DocXMLRPCServer)

Модуль `xmlrpc.server` содержит классы, предназначенные для реализации различных видов серверов XML-RPC. В Python 2 эта функциональность находится в двух отдельных модулях: `SimpleXMLRPCServer` и `DocXMLRPCServer`.

`SimpleXMLRPCServer(`*`addr`* `[, ` *`requestHandler`* `[, ` *`logRequests`*`]])`

Создает сервер XML-RPC, принимающий соединения по адресу *addr* (например, `('localhost',8080)`). В аргументе *requestHandler* передается фабрич-

ная функция, которая создает объекты для обработки запросов при подключении клиента. По умолчанию используется функция SimpleXMLRPCRequestHandler(), которая в настоящее время является конструктором единственного доступного обработчика. В аргументе *logRequests* передается логический флаг, который разрешает или запрещает журналирование входящих запросов. По умолчанию принимает значение True.

DocXMLRPCServer(*addr* [, *requestHandler* [, *logRequest*])

Создает описание сервера XML-RPC, которое возвращается в ответ на HTTP-запрос GET (обычно отправляемый броузерами). Когда сервер принимает такой запрос, он генерирует описание, объединяя все строки документирования, присутствующие во всех зарегистрированных методах и объектах. Аргументы этой функции имеют тот же смысл, что и в функции SimpleXMLRPCServer().

Экземпляр *s* класса SimpleXMLRPCServer или DocXMLRPCServer обладает следующими методами:

s.register_function(*func* [, *name*])

Регистрирует новую функцию *func* на сервере XML-RPC. Необязательный аргумент *name* определяет дополнительное имя функции. Если метод вызывается с аргументом *name*, клиенты будут использовать его значение для доступа к функции. Это имя может содержать символы, которые не допускается использовать в обычных идентификаторах языка Python, включая точки (.). Если аргумент *name* отсутствует, будет использоваться фактическое имя функции.

s.register_instance(*instance* [, *allow_dotted_names*])

Регистрирует объект, который используется для разрешения имен методов, не зарегистрированных с помощью метода register_function(). Если экземпляр *instance* определяет метод _dispatch(*self*, *methodname*, *params*), обработка запросов будет выполняться с его помощью. В аргументе *methodname* он принимает имя метода, а в аргументе *params* – кортеж с аргументами. Значение, возвращаемое методом _dispatch(), отправляется клиентам. Если метод _dispatch() не определен, проверяется наличие одноименного метода в самом экземпляре *instance*. Если такой метод определен, он вызывается. В аргументе *allow_dotted_names* передается флаг, разрешающий или запрещающий иерархический поиск имен методов. Например, если в запросе указан метод с именем 'foo.bar.spam', этот флаг определяет, должен ли выполняться поиск метода *instance*.foo.bar.spam. По умолчанию этот аргумент принимает значение False. Он не должен устанавливаться в значение True при работе с непроверенными клиентами. В противном случае он открывает брешь в системе безопасности, которая позволит злоумышленнику выполнить произвольный программный код на языке Python. Обратите внимание, что одним вызовом может быть зарегистрирован только один экземпляр.

s.register_introspection_functions()

Добавляет в сервер XML-RPC поддержку интроспекции с помощью функций system.listMethods(), system.methodHelp() и system.methodSignature().

Функция `system.methodHelp()` возвращает строку документирования для указанного метода (если имеется). Функция `system.methodSignature()` просто возвращает сообщение, в котором указывается, что данная операция не поддерживается (так как язык Python относится к языкам с динамической типизацией, информация о типах недоступна).

s`.register_multicall_functions()`

Добавляет в сервер **XML-RPC поддержку множественного вызова с помощью функции** `system.multicall()`.

Экземпляры класса `DocXMLRPCServer` дополнительно предоставляют следующие методы:

s`.set_server_title(`*server_title*`)`

Определяет заголовок в HTML-документации сервера. Строка *server_title* будет заключена в HTML-тег `<title>`.

s`.set_server_name(`*server_name*`)`

Определяет имя сервера в HTML-документации. Строка *server_name* будет помещена в начало страницы и заключена в тег `<h1>`.

s`.set_server_documentation(`*server_documentation*`)`

Добавляет к HTML-документации абзац с описанием. Содержимое *server_documentation* будет помещено сразу вслед за именем сервера, но перед описанием функций XML-RPC.

Обычно серверы XML-RPC выполняются как самостоятельные процессы, тем не менее имеется возможность запускать их внутри сценариев CGI. Для этого используются следующие классы:

`CGIXMLRPCRequestHandler([`*allow_none* `[, ` *encoding*`]])`

Обработчик запросов CGI, который действует так же, как экземпляр `SimpleXMLRPCServer`. Аргументы имеют тот же смысл, что и в конструкторе `SimpleXMLRPCServer()`.

`DocCGIXMLRPCRequestHandler()`

Обработчик запросов CGI, который действует так же, как экземпляр `DocXMLRPCServer`. Обратите внимание, что к моменту написания этих строк данный конструктор принимает иной набор аргументов, чем `CGIXMLRPCRequestHandler()`. Возможно, это ошибка, поэтому вам следует обращаться к электронной документации при использовании будущих версий.

Экземпляр *c* любого из классов обработчиков запросов CGI обладает теми же методами регистрации функций и объектов, что и обычный сервер XML-RPC. Но вдобавок он обладает следующим методом:

c`.handle_request([`*request_text*`])`

Обрабатывает запрос XML-RPC. По умолчанию запрос читается из потока стандартного ввода. Если методу передается аргумент *request_text*, он должен содержать данные в формате HTTP-запроса `POST`.

Примеры

Ниже приводится очень простой пример автономного сервера. Он добавляет единственную функцию add. Кроме того, он обеспечивает доступ ко всему содержимому модуля math, как к экземпляру класса, предоставляя возможность вызова любых функций, содержащихся в нем.

```
try:
    from xmlrpc.server import SimpleXMLRPCServer      # Python 3
except ImportError:
    from SimpleXMLRPCServer import SimpleXMLRPCServer # Python 2
import math

def add(x,y):
    "Складывает два числа "
    return x+y

s = SimpleXMLRPCServer(('',8080))
s.register_function(add)
s.register_instance(math)
s.register_introspection_functions()
s.serve_forever()
```

В следующем примере та же самая функциональность реализована в виде сценария CGI:

```
try:
    from xmlrpc.server import CGIXMLRPCRequestHandler      # Python 3
except ImportError:
    from SimpleXMLRPCServer import CGIXMLRPCRequestHandler # Python 2
import math

def add(x,y):
    "Складывает два числа"
    return x+y

s = CGIXMLRPCRequestHandler()
s.register_function(add)
s.register_instance(math)
s.register_introspection_functions()
s.handle_request()
```

Обратиться к функциям на сервере XML-RPC из других программ на языке Python можно с помощью модуля xmlrpc.client или xmlrpclib. Ниже приводится пример короткого интерактивного сеанса, который демонстрирует, как это делается:

```
>>> from xmlrpc.client import ServerProxy
>>> s = ServerProxy("http://localhost:8080")
>>> s.add(3,5)
8
>>> s.system.listMethods()
['acos', 'add', 'asin', 'atan', 'atan2', 'ceil', 'cos', 'cosh', 'degrees',
'exp', 'fabs', 'floor', 'fmod', 'frexp', 'hypot', 'ldexp', 'log', 'log10',
'modf', 'pow', 'radians', 'sin', 'sinh', 'sqrt', 'system.listMethods',
'system.methodHelp', 'system.methodSignature', 'tan', 'tanh']
```

```
>>> s.tan(4.5)
4.6373320545511847
>>>
```

Дополнительные возможности настройки сервера

Модули, содержащие реализацию серверов XML-RPC, удобно использовать для организации распределенных вычислений. Например, XML-RPC можно было бы использовать как протокол высокого уровня управления другими системами в сети, если во всех этих системах выполняются соответствующие серверы XML-RPC. С помощью модуля `pickle` можно организовать передачу более сложных объектов между системами.

Один из недостатков XML-RPC – низкий уровень безопасности. По умолчанию сервер XML-RPC действует, как открытая сетевая служба, поэтому любой, кому известен адрес и номер порта сервера, сможет подключиться к нему (если сервер не защищен брандмауэром). Кроме того, серверы XML-RPC не ограничивают объем данных, которые могут пересылаться в запросах. Вследствие этого злоумышленник может вызвать аварийное прекращение работы сервера, отправив HTTP-запрос, объем данных в котором превышает объем памяти, доступной серверу.

Если у вас появится потребность ликвидировать любую из этих проблем, это можно реализовать за счет переопределения классов серверов XML-RPC или обработчиков запросов. Все классы серверов являются производными от класса `TCPServer`, определение которого находится в модуле `socketserver`. То есть возможности серверов XML-RPC можно изменять точно так же, как возможности любых классов серверных сокетов (например, добавлять поддержку многопоточности или создания дочерних процессов, реализовать проверку адресов клиентов). В обработчики запросов можно добавить дополнительные проверки, создав класс, производный от `SimpleXMLRPCRequestHandler` или `DocXMLRPCRequestHandler`, и переопределив метод `do_POST()`. Ниже приводится пример, где добавлено ограничение на размер входящих запросов:

```
try:
    from xmlrpc.server import (SimpleXMLRPCServer,
                               SimpleXMLRPCRequestHandler)
except ImportError:
    from SimpleXMLRPCServer import (SimpleXMLRPCServer,
                                    SimpleXMLRPCRequestHandler)
class MaxSizeXMLRPCHandler(SimpleXMLRPCRequestHandler):
    MAXSIZE = 1024*1024   # 1 Мбайт
    def do_POST(self):
        size = int(self.headers.get('content-length',0))
        if size >= self.MAXSIZE:
            self.send_error(400,"Bad request")
        else:
            SimpleXMLRPCRequestHandler.do_POST(self)

s = SimpleXMLRPCServer(('',8080),MaxSizeXMLRPCHandler)
```

Если потребуется добавить механизм аутентификации на основе протокола HTTP, ее также можно реализовать подобным способом.

23

Веб-программирование

Язык программирования Python широко используется при построении веб-сайтов и применяется для решения самых разных задач. Во-первых, сценарии на языке Python часто обеспечивают простой и удобный способ генерирования набора статических HTML-страниц, которые будут обслуживаться веб-сервером. Например, сценарий может принимать некоторое содержимое и добавлять к нему элементы оформления, типичные для веб-сайта (панель навигации, боковую панель, рекламу, стили и так далее). Все это фактически сводится к работе с файлами и обработке текста, о чем рассказывается в других разделах книги.

Во-вторых, сценарии на языке Python могут использоваться для создания динамического содержимого. Например, веб-сайт может работать под управлением стандартного веб-сервера, такого как Apache, и использовать сценарии на языке Python для динамической обработки некоторых видов запросов. В этом случае использование Python сводится к обработке форм. Например, страница HTML может включать форму, как показано ниже:

```
<FORM ACTION='/cgi-bin/subscribe.py' METHOD='GET'>
Your name : <INPUT type='Text' name='name' size='30'>
Your email address: <INPUT type='Text' name='email' size='30'>
<INPUT type='Submit' name='submit-button' value='Subscribe'>
</FORM>
```

Атрибут `ACTION` внутри формы определяет имя сценария на языке Python `'subscribe.py'`, который будет выполнен сервером при получении формы.

Также достаточно часто динамическое содержимое генерируется с использованием технологии AJAX (Asynchronous JavaScript and XML – асинхронный JavaScript и XML). Эта технология основана на создании обработчиков событий на JavaScript для определенных элементов HTML-страниц. Например, когда пользователь помещает указатель мыши над определенным элементом документа, функция на JavaScript может оправить HTTP-запрос веб-серверу, чтобы обработать некоторые данные (возможно с помощью сценария на языке Python). После получения ответа другая функция

на JavaScript может обработать полученные данные и отобразить результаты. Результаты могут возвращаться сервером в самых разных форматах. Например, сервер может возвращать данные в виде простого текста, XML, JSON или в любых других форматах. Ниже приводится пример документа HTML, в котором иллюстрируется один из способов реализации отображения всплывающего окна, когда указатель мыши оказывается над выделенными элементами.

```
<html>
  <head>
    <title>ACME Officials Quiet After Corruption Probe</title>
    <style type="text/css">
      .popup { border-bottom:1px dashed green; }
      .popup:hover { background-color: #c0c0ff; }
    </style>
  </head>
  <body>
    <span id="popupbox"
      style="visibility:hidden; position:absolute; background-color:#ffffff;">
      <span id="popupcontent"></span>
    </span>
    <script>
      /* Получить ссылку на всплывающее окно */
      var popup = document.getElementById("popupbox");
      var popupcontent = document.getElementById("popupcontent");

      /* Получает от сервера данные и выводит их во всплывающем окне */
      function ShowPopup(hoveritem,name) {
        var request = new XMLHttpRequest();
        request.open("GET","cgi-bin/popupdata.py?name="+name, true);
        request.onreadystatechange = function() {
          var done = 4, ok = 200;
          if (request.readyState == done && request.status == ok) {
            if (request.responseText) {
              popupcontent.innerHTML = request.responseText;
              popup.style.left = hoveritem.offsetLeft+10;
              popup.style.top = hoveritem.offsetTop+20;
              popup.style.visibility = "Visible";
            }
          }
        };
        request.send();
      }

      /* Скрывает всплывающее окно */
      function HidePopup() {
        popup.style.visibility = "Hidden";
      }
    </script>
    <h3>ACME Officials Quiet After Corruption Probe</h3>
    <p>
    Today, shares of ACME corporation
   (<span class="popup" onMouseOver="ShowPopup(this,'ACME');"
```

```
        onMouseOut="HidePopup();">ACME</span>)
        plummetted by more than 75% after federal investigators revealed that
        the board of directors is the target of a corruption probe involving
        the Governor, state lottery officials, and the archbishop.
        </p>
      </body>
    </html>
```

В этом примере функция `ShowPopup()` на JavaScript инициирует запрос к сценарию на языке Python `popupdata.py`, находящемуся на сервере. Результатом работы этого сценария является всего лишь фрагмент разметки HTML, который отображается во всплывающем окне. На рис. 23.1 показано, как это выглядит в окне броузера.

Наконец, весь веб-сайт целиком может работать под управлением Python, в контексте фреймворка, написанного на языке Python. Как было шутливо отмечено: «количество фреймворков, написанных на языке Python, превышает количество ключевых слов языка». Тема построения веб-фреймворков выходит далеко за рамки этой книги, но тем, кому это интересно, можно порекомендовать начать изучение этого вопроса с посещения страницы *http://wiki.python.org/moin/WebFrameworks*.

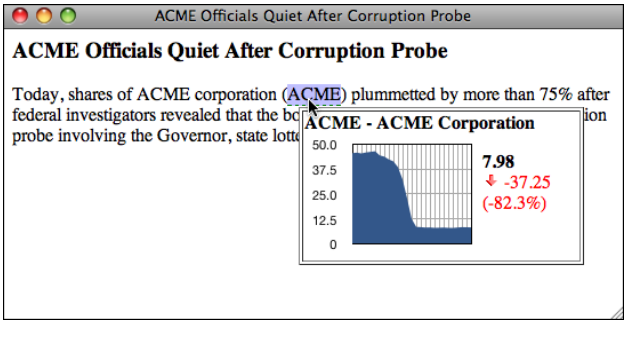


Рис. 23.1. *Так в окне броузера выглядит текст обычного документа HTML и всплывающее окно с динамическим содержимым, созданным с помощью сценария popupdata.py*

В оставшейся части главы описываются встроенные модули, реализующие низкоуровневый интерфейс для взаимодействия с веб-серверами и фреймворками. В число рассматриваемых тем входят: создание сценариев CGI, приемы использования сценариев на языке Python в сторонних веб-серверах с применением стандарта WSGI, создание компонентов промежуточного уровня, которые интегрируются с различными веб-фреймворками на языке Python.

Модуль cgi

Модуль `cgi` используется для реализации *сценариев CGI*, или программ, которые обычно вызываются веб-сервером для обработки данных формы или для создания некоторого динамического содержимого.

Когда отправляется запрос к соответствующему сценарию CGI, веб-сервер запускает CGI-программу в отдельном дочернем процессе. Программы CGI получают входные данные из двух источников: из потока `sys.stdin` и из переменных окружения, устанавливаемых сервером. Ниже перечислены наиболее типичные переменные окружения, которые устанавливаются веб-сервером:

Переменная	Описание
`AUTH_TYPE`	Метод аутентификации
`CONTENT_LENGTH`	Длина переданных данных в `sys.stdin`
`CONTENT_TYPE`	Тип данных запроса
`DOCUMENT_ROOT`	Корневой каталог с документами
`GATEWAY_INTERFACE`	Строка с номером версии CGI
`HTTP_ACCEPT`	Типы MIME, принимаемые клиентом
`HTTP_COOKIE`	Хранилище значения cookie
`HTTP_FROM`	Адрес электронной почты клиента (часто отправка этой информации бывает запрещена)
`HTTP_REFERER`	Ссылающийся адрес URL
`HTTP_USER_AGENT`	Информация о броузере клиента
`PATH_INFO`	Дополнительная информация о пути
`PATH_TRANSLATED`	Преобразованная версия переменной `PATH_INFO`
`QUERY_STRING`	Строка запроса
`REMOTE_ADDR`	IP-адрес клиента
`REMOTE_HOST`	Имя хоста клиента
`REMOTE_IDENT`	Пользователь, выполнивший запрос
`REMOTE_USER`	Имя аутентифицированного пользователя
`REQUEST_METHOD`	Метод (`'GET'` или `'POST'`)
`SCRIPT_NAME`	Имя программы
`SERVER_NAME`	Имя хоста сервера
`SERVER_PORT`	Номер порта сервера
`SERVER_PROTOCOL`	Протокол сервера
`SERVER_SOFTWARE`	Название и версия серверного программного обеспечения

Вывод программы CGI записывается в поток стандартного вывода `sys.stdout`. Технические подробности, касающиеся разработки CGI-сценариев, можно найти, например, в книге «CGI Programming with Perl, 2nd Edition»,[1]

[1] Гундаварам Ш. «CGI-программирование на Perl, 2-е издание». – Пер с англ. – СПб.: Символ-Плюс, 2001.

Шишира Гундаварама (Shishir Gundavaram) (O'Reilly, 2000). Нам же достаточно знать два основных аспекта. Во-первых, содержимое формы HTML передается CGI-программе в виде текстовой последовательности, которая называется *строкой запроса*. В языке Python доступ к содержимому строки запроса осуществляется с помощью класса `FieldStorage`. Например:

```
import cgi
form = cgi.FieldStorage()
name = form.getvalue('name')   # Получить содержимое поля 'name' формы
email = form.getvalue('email') # Получить содержимое поля 'email' формы
```

Во-вторых, вывод CGI-программы делится на две части: заголовок HTTP и собственно данные (обычно в формате HTML). Эти две части всегда отделяются друг от друга пустой строкой. Ниже показано, как производится вывод простого заголовка HTTP:

```
print 'Content-type: text/html\r'   # Данные будут выводиться в формате HTML
print '\r'                          # Пустая строка (обязательно!)
```

Остальная часть вывода – это собственно данные. Например:

```
print '<TITLE>My CGI Script</TITLE>'
print '<H1>Hello World!</H1>'
print 'You are %s (%s)' % (name, email)
```

Стало стандартной практикой завершать заголовки HTTP последовательностью символов `'\r\n'` завершения строки в стиле Windows. Именно по этой причине в примере выполняется вывод символа `'\r'`. Если программе потребуется сообщить об ошибке, можно добавить вывод специального заголовка `'Status:'`. Например:

```
print 'Status: 401 Forbidden\r'          # Код ошибки HTTP
print 'Content-type: text/plain\r'
print '\r'                               # Пустая строка (обязательно)
print 'You're not worthy of accessing this page!'
```

Если необходимо переадресовать клиента на другую страницу, можно создать такой вывод:

```
print 'Status: 302 Moved\r'
print 'Location: http://www.foo.com/orderconfirm.html\r'
print '\r'
```

Основная работа выполняется модулем `cgi` при создании экземпляра класса `FieldStorage`.

`FieldStorage([`*`input`* `[, `*`headers`* `[, `*`outerboundary`* `[, `*`environ`* `[, `*`keep_blank_values`* `[, `*`strict_parsing`*`]]]]]])`

Читает содержимое формы, анализируя строку запроса, переданную через переменную окружения или через поток стандартного ввода. Аргумент *input* определяет объект, похожий на файл, из которого будет выполняться чтение данных формы, полученных в составе запроса `POST`. По умолчанию используется объект `sys.stdin`. Аргументы *headers* и *outerboundary* использу-

ются для внутренних нужд и не должны указываться в вызове конструктора. В аргументе *environ* передается словарь, содержащий переменные окружения для CGI-сценария. В аргументе *keep_blank_values* передается логический флаг, разрешающий сохранение пустых значений. По умолчанию принимает значение False. В аргументе *strict_parsing* передается логический флаг, разрешающий возбуждение исключения в случае обнаружения ошибок при анализе. По умолчанию принимает значение False.

Экземпляр *form* класса FieldStorage действует как словарь. Например, инструкция *f* = *form*[*key*] извлечет информацию о параметре *key*. Экземпляр *f*, полученный таким способом, будет либо еще одним экземпляром класса FieldStorage, либо экземпляром класса MiniFieldStorage. Ниже перечислены атрибуты, которыми обладает экземпляр *f*:

Атрибут	Описание
f.name	Имя поля, если указано
f.filename	Имя выгружаемого файла на стороне клиента
f.value	Значение в виде строки
f.file	Объект, похожий на файл, с помощью которого можно читать данные
f.type	Тип содержимого
f.type_options	Словарь с параметрами, указанными в заголовке content-type HTTP-запроса
f.disposition	Значение заголовка 'content-disposition'; None – если не указан
f.disposition_options	Словарь с параметрами, указанными в заголовке 'content-disposition
f.headers	Объект, похожий на словарь, содержащий все заголовки HTTP

Значения полей формы могут извлекаться с помощью следующих методов:

form.getvalue(*fieldname* [, *default*])

Возвращает значение поля с именем *fieldname*. Если поле определено дважды, этот метод вернет список всех значений. Аргумент *default* определяет значение, возвращаемое в случае отсутствия указанного поля. Будьте внимательны: если одно и то же имя поля включено в запрос дважды, этот метод вернет список, содержащий оба значения. Чтобы упростить работу с такими значениями, можно использовать метод *form*.getfirst(), который возвращает первое найденное значение.

form.getfirst(*fieldname* [, *default*])

Возвращает первое значение поля с именем *fieldname*. Аргумент *default* определяет значение, возвращаемое в случае отсутствия указанного поля.

form.getlist(*fieldname*)

Возвращает список всех значений поля с именем *fieldname*. Этот метод всегда возвращает список, даже если поле имеет единственное значение. В случае отсутствия значений возвращает пустой список.

Дополнительно модуль cgi объявляет класс MiniFieldStorage, содержащий только атрибуты name и value. Этот класс используется для представления отдельных полей формы, переданных в строке запроса, тогда как класс FieldStorage используется для представления набора полей и для представления полей с несколькими значениями.

Экземпляры класса FieldStorage действуют, как словари Python, в которых ключами являются имена полей формы. При обращении к ним таким способом они возвращают объекты, которые сами являются экземплярами класса FieldStorage — в случае полей с несколькими значениями (тип содержимого 'multipart/form-data') или полей с выгружаемыми файлами. В случае простых полей (тип содержимого 'application/x-www-form-urlencoded') возвращаются экземпляры класса MiniFieldStorage или списки экземпляров этого класса, если форма содержит несколько полей с одинаковыми именами. Например:

```
form = cgi.FieldStorage()
if "name" not in form:
    error("Name is missing")
    return
name = form['name'].value   # Вернет поле 'name' формы
email = form['email'].value # Вернет поле 'email' формы
```

Если поле представляет выгружаемый файл, операция обращения к атрибуту value прочитает все содержимое файла и сохранит его в памяти в виде строки байтов. Так как это может приводить к потреблению существенных объемов памяти на стороне сервера, предпочтительнее читать содержимое файла небольшими фрагментами, с помощью атрибута file. Например, в следующем примере демонстрируется построчное чтение выгружаемого файла:

```
fileitem = form['userfile']
if fileitem.file:
    # Это выгружаемый файл; подсчитать строки
    linecount = 0
    while True:
        line = fileitem.file.readline()
        if not line: break
        linecount = linecount + 1
```

Ниже перечислены вспомогательные функции, которые часто используются в CGI-сценариях:

escape(*s* [, *quote*])

Преобразует символы '&', '< и '>', присутствующие в строке *s*, в соответствующие сущности языка разметки HTML, такие как '&', '<' и '>'.

Если в необязательном булевском аргументе *quote* передается истинное значение, символ двойной кавычки (") также замещается сущностью '"'.

`parse_header(string)`

Анализирует данные, следующие за именем HTTP-заголовка, такого как 'content-type'. Возвращает кортеж, содержащий основное значение и словарь с параметрами. Например, вызов

```
parse_header('text/html; a=hello; b="world"')
```

вернет:

```
('text/html', {'a':'hello', 'b':'world'}).
```

`parse_multipart(fp, pdict)`

Анализирует входные данные типа 'multipart/form-data', которые обычно представляют выгружаемый файл. В аргументе *fp* передается входной файл, а в аргументе *pdict* – словарь с параметрами из заголовка 'content-type'. Возвращает словарь, отображающий имена полей в списки значений. Эта функция не обрабатывает вложенные части 'multipart/*'. Для работы с такими полями следует использовать класс FieldStorage.

`print_directory()`

Преобразует имя текущего рабочего каталога в формат HTML и выводит его. Вывод отправляется обратно броузеру клиента. Может использоваться для отладки.

`print_environ()`

Создает список всех переменных окружения в формате HTML. Может использоваться для отладки.

`print_environ_usage()`

Выводит список наиболее полезных переменных окружения в формате HTML. Может использоваться для отладки.

`print_form(form)`

Форматирует данные, полученные в составе формы. В аргументе *form* должен передаваться экземпляр класса FieldStorage. Может использоваться для отладки.

`test()`

Выводит минимальный HTTP-заголовок и всю переданную программе информацию в формате HTML. В основном используется для отладки, позволяя убедиться, что окружение CGI настроено корректно.

Советы по созданию CGI-сценариев

При нынешнем уровне развития веб-фреймворков разработка CGI-сценариев выходит из моды. Однако если у вас появится потребность в их создании, следующие несколько советов помогут вам упростить эту работу.

Во-первых, не используйте большое количество инструкций print для вывода жестко определенной разметки HTML. В противном случае програм-

ма превратится в жуткую смесь инструкций языка Python и разметки HTML, которую не только невозможно читать, но и невозможно поддерживать. Лучше всего использовать шаблоны. Для этого можно использовать объекты `string.Template`. Ниже приводится пример, демонстрирующий использование этого подхода:

```
import cgi
from string import Template

def error(message):
    temp = Template(open("errormsg.html").read())
    print 'Content-type: text/html\r'
    print '\r'
    print temp.substitute({'message' : message})

form = cgi.FieldStorage()
name = form.getfirst('name')
email = form.getfirst('email')
if not name:
    error("name not specified")
    raise SystemExit
elif not email:
    error("email not specified")
    raise SystemExit

# Выполнить обработку данных
confirmation = subscribe(name, email)

# Вывести содержимое страницы
values = {
    'name' : name,
    'email' : email,
    'confirmation: ': confirmation,
    # Здесь можно добавить другие значения...
}
temp = Template(open("success.html").read()
print temp.substitute(values)
```

В этом примере используются файлы `'error.html'` и `'success.html'`, которые содержат всю необходимую разметку HTML, но включают символы подстановки вида $*variable*, соответствующие значениям, генерируемым в CGI-сценарии. Например, содержимое файла `'success.html'` могло бы выглядеть примерно так:

```
<HTML>
    <HEAD>
        <TITLE>Success</TITLE>
    </HEAD>
    <BODY>
        Welcome $name. You have successfully subscribed to our
        newsletter. Your confirmation code is $confirmation.
    </BODY>
</HTML>
```

Вызов метода `temp.substitute()` в сценарии просто замещает символы подстановки в этом файле. Очевидное преимущество этого подхода в том, что при необходимости изменить внешний вид страницы достаточно будет изменить файл шаблона, не изменяя сам CGI-сценарий. Для языка Python существует множество сторонних реализаций механизмов управления шаблонами; возможно, их даже больше, чем веб-фреймворков. Все они в значительной степени основываются на понятии шаблона. Дополнительные подробности вы найдете на странице *http://wiki.python.org/moin/Templating*.

Во-вторых, если имеется необходимость хранения данных для CGI-сценариев, старайтесь использовать базу данных. Несомненно, запись данных прямо в файлы выполняется достаточно просто, однако не забывайте, что веб-серверы действуют в многозадачной среде, и если не предпринять дополнительных усилий по синхронизации доступа к ресурсам, велика вероятность повредить эти файлы. Серверы баз данных и соответствующие интерфейсы доступа к ним из программ на языке Python обычно лишены этого недостатка. Поэтому если вам понадобится сохранять данные, попробуйте задействовать модуль, такой как `sqlite3`, или сторонний модуль доступа к базам данных, например к MySQL.

Наконец, если вы вдруг обнаружите, что приходится писать десятки CGI-сценариев, в которых приходится заниматься обработкой низкоуровневых особенностей протокола **HTTP, таких как cookie, аутентификация, кодировка символов** и так далее, подумайте об использовании веб-фреймворка. Основная цель фреймворка состоит в том, чтобы избавить вас от необходимости задумываться о таких деталях или, по крайней мере, от необходимости глубоко в них погружаться. Не старайтесь повторно изобрести колесо.

Примечания

- Процедура установки CGI-программы может существенно отличаться, в зависимости от типа используемого веб-сервера. Обычно CGI-программы помещаются в специальный каталог `cgi-bin`. Кроме того, может потребоваться выполнить дополнительные настройки сервера. За дополнительной информацией вам следует обратиться к документации с описанием сервера или к администратору сервера.
- В системе **UNIX может потребоваться присвоить CGI-программам на языке Python соответствующие разрешения на выполнение, а сама программа** должна будет начинаться со строки, аналогичной показанной ниже:

```
#!/usr/bin/env python
import cgi
...
```

  Чтобы упростить отладку, импортируйте модуль `cgitb`. Например, добавьте инструкции `import cgitb; cgitb.enable()`. Это изменит обработку исключений так, что все сообщения об ошибках будут отображаться в веб-броузере.

- Если **CGI-сценарий вызывает внешнюю программу, например с помощью функции** `os.system()` или `os.popen()`, будьте особенно внимательны

ны и не передавайте командной оболочке непроверенные строки, полученные от пользователя. Это хорошо известная прореха в системе безопасности, которая может использоваться злоумышленниками для выполнения на сервере произвольных системных команд (потому что команды, передаваемые этим функциям, интерпретируются командной оболочкой UNIX). В частности, никогда не передавайте командной оболочке какие-либо части строки **URL или данные из формы, предварительно не проверив их и не убедившись**, что строки содержат только алфавитно-цифровые символы, дефисы, символы подчеркивания и точки.

- В системе UNIX никогда не устанавливайте бит `setuid` для файлов CGI-программ. Это может привести к проблемам с безопасностью, а кроме того, такая возможность поддерживается не во всех системах.
- Не импортируйте этот модуль инструкцией '`from cgi import *`'. Модуль `cgi` объявляет огромное количество символов, которыми не стоит забивать пространство имен программы.

Модуль cgitb

Этот модуль предоставляет альтернативную реализацию механизма обработки исключений, которая отображает подробные отчеты обо всех необработанных исключениях. В отчет включается исходный программный код, значения параметров и локальных переменных. Первоначально этот модуль создавался с целью помочь в отладке CGI-сценариев, но он точно так же может использоваться любыми другими приложениями.

`enable([`*`display`* `[, `*`logdir`* `[, `*`context`* `[, `*`format`*`]]]])`

Разрешает специализированную обработку исключений. В аргументе *display* передается флаг, разрешающий вывод информации в случае появления ошибки. По умолчанию принимает значение `1`. Аргумент *logdir* определяет каталог, куда должны сохраняться файлы с отчетами, вместо вывода в поток стандартного вывода. Если аргумент *logdir* определен, каждый отчет будет сохранен в отдельном файле, созданном с помощью функции `tempfile.mkstemp()`. В аргументе *context* передается целое число, определяющее количество строк с исходным программным кодом, окружающим строку, где возникло исключение. Аргумент *format* определяет выходной формат отчетов. Значение '`html`' (по умолчанию) в аргументе *format* соответствует формату HTML. Если указать любое другое значение, отчеты будут выводиться в простом текстовом формате.

`handle([`*`info`*`])`

Обрабатывает исключение, используя настройки по умолчанию функции `enable()`. В аргументе *info* передается кортеж (*exctype, excvalue, tb*), где поле *exctype* представляет тип исключения, *excvalue* – значение исключения, а поле *tb* – объект с трассировочной информацией. Этот кортеж обычно можно получить вызовом функции `sys.exc_info()`. Если аргумент *info* опущен, используется информация о текущем исключении.

Примечание

Чтобы включить специальный режим обработки исключений в CGI-сценариях, добавьте в начало сценария строку `import cgitb; enable()`.

Поддержка WSGI

WSGI (Web Server Gateway Interface – шлюзовой интерфейс веб-сервера) – это стандартизованный интерфейс между веб-серверами и веб-приложениями, который был разработан для обеспечения независимости приложений от типов веб-серверов и фреймворков. Официальное описание стандарта можно найти в документе PEP 333 (*http://www.python.org/dev/peps/pep-0333*). Дополнительные сведения о стандарте и его использовании можно также найти на сайте *http://www.wsgi.org*. Пакет `wsgiref` содержит реализацию вспомогательных функций, которые могут использоваться для тестирования, проверки и упрощения развертывания.

Спецификация стандарта WSGI

Согласно WSGI, **веб-приложение реализуется как функция, или вызываемый объект,** *webapp*(*environ, start_response*), которая принимает два аргумента, где аргумент *environ* – это словарь с параметрами окружения, который должен, как минимум, содержать следующие значения, имеющие те же имена и то же назначение, что и в CGI-сценариях:

Переменная	Описание
CONTENT_LENGTH	Длина переданных данных
CONTENT_TYPE	Тип данных запроса
HTTP_ACCEPT	Типы MIME, принимаемые клиентом
HTTP_COOKIE	Хранилище значения cookie
HTTP_REFERER	Ссылающийся адрес URL
HTTP_USER_AGENT	Информация о броузере клиента
PATH_INFO	Дополнительная информация о пути
QUERY_STRING	Строка запроса
REQUEST_METHOD	Метод ('GET' или 'POST')
SCRIPT_NAME	Имя программы
SERVER_NAME	Имя хоста сервера
SERVER_PORT	Номер порта сервера
SERVER_PROTOCOL	Протокол сервера

Кроме того, словарь *environ* должен содержать следующие значения, характерные для WSGI:

Переменная	Описание
wsgi.version	Кортеж, представляющий версию WSGI (например, (1,0) для WSGI 1.0).
wsgi.url_scheme	Строка, содержащая компонент адреса URL с названием протокола. Например, 'http' или 'https'.
wsgi.input	Объект, похожий на файл, представляющий поток стандартного ввода. Из этого потока будут читаться дополнительные данные, такие как данные формы или выгружаемый файл.
wsgi.errors	Объект, похожий на файл, открытый в текстовом режиме и используемый для вывода сообщений об ошибках.
wsgi.multithread	Логический флаг, значение True в котором разрешает запустить приложение в отдельном потоке управления в рамках одного и того же процесса.
wsgi.multiprocess	Логический флаг, значение True в котором разрешает запустить приложение в дочернем процессе процессов.
wsgi.run_once	Логический флаг; значение True в котором говорит о том, что приложение может запускаться лишь один раз в течение всего времени выполнения процесса.

В аргументе *start_response* передается вызываемый объект вида start_response(*status*, *headers*), который используется приложением для запуска ответа. В аргументе *status* этот объект принимает строку с кодом ответа, такую как '200 OK' или '404 Not Found', а в аргументе *headers* – список кортежей вида (*name*, *value*), соответствующих HTTP-заголовкам, включаемым в ответ, например ('Content-type','text/html').

Данные, или тело ответа, возвращаются функцией веб-приложения в виде итерируемого объекта, воспроизводящего последовательность текстовых строк или строк байтов, содержащих только символы, которые могут быть представлены единственным байтом (например, совместимые с набором символов ISO-8859-1 или Latin-1). Это может быть список строк байтов или функция-генератор, воспроизводящая строки байтов. Если приложению необходимо выполнить кодирование символов, например, в кодировку UTF-8, оно должно делать это самостоятельно.

Ниже приводится пример простого приложения WSGI, которое читает значения полей формы и производит вывод, подобный тому, что был показан в разделе с описанием модуля cgi:

```
import cgi
def subscribe_app(environ, start_response):
    fields = cgi.FieldStorage(environ['wsgi.input'],
                              environ=environ)

    name = fields.getvalue("name")
    email = fields.getvalue("email")

    # Обработка формы
```

```
    status = "200 OK"
    headers = [('Content-type','text/plain')]
    start_response(status, headers)

    response = [
        'Hi %s. Thank you for subscribing.' % name,
        'You should expect a response soon.'
        ]
    return (line.encode('utf-8') for line in response)
```

В этом примере имеется несколько важных моментов. Во-первых, компоненты WSGI-приложения не привязаны к какому-либо определенному фреймворку, веб-серверу или набору библиотечных модулей. Здесь использовался только один библиотечный модуль — `cgi`, и то лишь потому, что он содержит ряд удобных функций, позволяющих анализировать содержимое строки запроса. Пример демонстрирует, как использовать функцию `start_response()`, чтобы инициировать ответ и применить заголовки. Сам ответ конструируется как список строк. Последняя инструкция в этом приложении создает выражение-генератор, преобразующее все строки в строки байтов. В Python 3 это очень важный шаг – все WSGI-приложения должны возвращать кодированные строки байтов, а не строки Юникода.

Чтобы ввести WSGI-приложение в эксплуатацию, его необходимо зарегистрировать средствами используемого фреймворка. А для этого следует ознакомиться с соответствующим руководством.

Пакет wsgiref

Пакет `wsgiref` содержит реализацию стандарта WSGI, позволяющую тестировать приложения на автономных серверах или выполнять их, как обычные CGI-сценарии.

Модуль wsgiref.simple_server

Модуль `wsgiref.simple_server` реализует простой автономный HTTP-сервер, выполняющий единственное WSGI-приложение. В этом модуле существуют всего две функции, представляющие для нас интерес:

`make_server(`*`host`*`, `*`port`*`, `*`app`*`)`

Создает HTTP-сервер, принимающий соединения по указанному адресу. Аргумент *host* определяет имя хоста, а аргумент *port* — номер порта. В аргументе *app* передается функция, или вызываемый объект, реализующая WSGI-приложение. Чтобы запустить сервер, необходимо вызвать метод *s*`.serve_forever()`, где *s* — это экземпляр сервера, возвращаемого функцией `make_server()`.

`demo_app(`*`environ`*`, `*`start_response`*`)`

Полноценное WSGI-приложение, возвращающее страницу с сообщением «Hello World». Может использоваться в качестве аргумента *app* функции `make_server()` для проверки работоспособности сервера.

Ниже приводится пример запуска простого WSGI-сервера:

```
def my_app(environ, start_response):
    # Некоторое приложение
    start_response("200 OK", [('Content-type','text/plain')])
    return ['Hello World']

if __name__ == '__main__':
    from wsgiref.simple_server import make_server
    serv = make_server('',8080, my_app)
    serv.serve_forever()
```

Модуль wsgiref.handlers

Модуль `wsgiref.handlers` содержит определения классов обработчиков, предназначенных для настройки среды выполнения WSGI, чтобы приложения могли выполняться под управлением другого веб-сервера (например, в виде CGI-сценария под управлением веб-сервера Apache). В модуле определено несколько разных классов.

`CGIHandler()`

Создает объект-обработчик WSGI, способный выполняться в стандартном окружении CGI. Этот обработчик извлекает информацию из стандартных переменных окружения и потоков ввода-вывода, о которых рассказывалось в описании модуля `cgi`.

`BaseCGIHandler(`*`stdin`*`, `*`stdout`*`, `*`stderr`*`, `*`environ`*` [, `*`multithread`*` [, `*`multiprocess`*`]])`

Создает объект-обработчик WSGI, способный выполняться в окружении CGI, но позволяет переопределять стандартные потоки ввода-вывода и переменные окружения. В аргументах *`stdin`*, *`stdout`* и *`stderr`* передаются объекты, похожие на файлы, которые будут использоваться как стандартные потоки ввода-вывода. В аргументе *`environ`* передается словарь с переменными окружения, который должен содержать стандартные переменные окружения CGI. В аргументах *`multithread`* и *`multiprocess`* передаются логические флаги, значения которых будут использоваться для установки значений переменных окружения `wsgi.multithread` и `wsgi.multiprocess`. По умолчанию аргумент *`multithread`* принимает значение `True`, а аргумент *`multiprocess`* — значение `False`.

`SimpleHandler(`*`stdin`*`, `*`stdout`*`, `*`stderr`*`, `*`environ`*` [, `*`multithread`*` [, `*`multiprocess`*`]])`

Создает объект-обработчик WSGI, похожий на объект класса `BaseCGIHandler`, но в отличие от последнего обеспечивающий приложению прямой доступ к аргументам *`stdin`*, *`stdout`*, *`stderr`* и *`environ`*. Объекты этого типа немного отличаются от объектов класса `BaseCGIHandler`, который предоставляет дополнительную логику, обеспечивающую корректную работу некоторых особенностей (например, объекты класса `BaseCGIHandler` преобразуют коды ответов в заголовки `Status:`).

Все эти обработчики обладают методом `run(`*`app`*`)`, который используется для запуска WSGI-приложения в окружении обработчика. Ниже приводится пример WSGI-приложения, которое запускается как обычный CGI-сценарий:

```
#!/usr/bin/env python
def my_app(environ, start_response):
    # Некоторое приложение
    start_response("200 OK",[('Content-type','text/plain')])
    return ['Hello World']

from wgiref.handlers import CGIHandler
hand = CGIHandler()
hand.run(my_app)
```

Модуль wsgiref.validate

Модуль `wsgiref.validate` содержит функции, позволяющие обертывать WSGI-приложения и проверять соответствие стандарту самого приложения и сервера, под управлением которого оно выполняется.

`validator(`*app*`)`

Создает новое WSGI-приложение, обертывающее WSGI-приложение *app*. Новое приложение действует точно так же, как и оригинальное приложение *app*, за исключением того, что к нему добавляются многочисленные проверки на наличие ошибок, позволяющие гарантировать, что само приложение и сервер соответствуют требованиям стандарта WSGI. Любое несоответствие вызывает появление исключения `AssertionError`.

Ниже приводится пример использования функции `validator()`:

```
def my_app(environ, start_response):
    # Некоторое приложение
    start_response("200 OK",[('Content-type','text/plain')])
    return ['Hello World']

if __name__ == '__main__':
    from wsgiref.simple_server import make_server
    from wsgiref.validate import validator
    serv = make_server('',8080, validator(my_app))
    serv.serve_forever()
```

Примечание

Сведения в этом разделе в первую очередь предназначены пользователям, которым требуется создавать объекты WSGI-приложений. Если же вы собираетесь реализовать свой фреймворк на языке Python, вам следует обратиться к документу PEP 333, где приводится официальная информация о том, какие функции должны быть реализованы во фреймворке, поддерживающем стандарт WSGI. В случае использования готовых фреймворков сторонних разработчиков обращайтесь к документации с описанием фреймворка за конкретной информацией о поддержке фреймворком объектов WSGI. Учитывая, что в стандартной библиотеке реализована отличная поддержка официальной спецификации WSGI, для фреймворков становится все более типичным обеспечивать определенный уровень поддержки этого стандарта.

Модуль webbrowser

Модуль webbrowser содержит вспомогательные функции, позволяющие открывать документы в веб-броузере платформонезависимым способом. В основном этот модуль используется в процессе разработки и тестирования. Например, при написании сценария, который воспроизводит разметку HTML, с помощью функций из этого модуля можно было бы автоматически вызывать броузер, используемый в системе по умолчанию, для просмотра результатов.

`open(`*`url`* `[, `*`new`* `[, `*`autoraise`*`]])`

Отображает страницу с адресом *url* в броузере по умолчанию. Если в аргументе *new* передать значение 0, страница будет открыта в открытом окне броузера, если это возможно. Если в аргументе *new* передать значение 1, будет открыто новое окно броузера. Если в аргументе *new* передать значение 2, страница будет открыта в новой вкладке броузера. Если в аргументе *autoraise* передать значение True, окно броузера будет выведено на передний план.

`open_new(`*`url`*`)`

Отображает страницу с адресом *url* в новом окне броузера по умолчанию. Идентична вызову open(*url*, 1).

`open_new_tab(`*`url`*`)`

Отображает страницу с адресом *url* в новой вкладке броузера по умолчанию. Идентична вызову open(*url*, 2).

`get([`*`name`*`])`

Возвращает объект управления броузером. В аргументе *name* передается имя броузера, которое представлено строкой, такой как 'netscape', 'mozilla', 'kfm', 'grail', 'windows-default', 'internet-config' или 'command-line'. Возвращаемый объект обладает методами open() и open_new(), которые принимают те же аргументы и выполняют те же действия, что и две предыдущие функции. При вызове без аргумента *name* возвращает объект управления броузером по умолчанию.

`register(`*`name`*`, `*`constructor`* `[, `*`controller`*`])`

Регистрирует новый тип броузера для использования в вызове функции get(). В аргументе *name* передается имя броузера. В аргументе *constructor* передается функция, которая вызывается без аргументов, – для создания объекта управления броузером. В аргументе *controller* передается готовый объект управления. Если этот аргумент определен, аргумент *constructor* игнорируется и может иметь значение None.

Экземпляр объекта управления *c*, который возвращается функцией get(), обладает следующими методами:

c`.open(`*`url`* `[, `*`new`*`])`

То же, что и функция open().

`c.open_new(`*`url`*`)`

То же, что и функция open_new().

24

Обработка и представление данных в Интернете

В этой главе описываются модули, обеспечивающие поддержку форматов, таких как base 64, HTML, XML и JSON, обычно используемых для передачи данных через Интернет.

Модуль base64

Модуль `base64` используется для преобразования двоичных данных в текстовое представление и обратно, с использованием кодировки base 64, base 32 или base 16. Кодировка base 64 обычно используется для кодирования двоичных данных, пересылаемых по электронной почте, в виде вложений, или по протоколу HTTP. Официальное описание этого формата приводится в документах RFC-3548 и RFC-1421.

При использовании формата base 64 данные объединяются в группы по 24 бита (3 байта). Затем каждая 24-битная группа делится на четыре 6-битовых компонента. Далее каждое 6-битное значение представляется в виде отдельного печатного символа ASCII из следующего алфавита:

Значение	**Символ**
0-25	ABCDEFGHIJKLMNOPQRSTUVWXYZ
26-51	abcdefghijklmnopqrstuvwxyz
52-61	0123456789
62	+
63	/
дополнение	=

Если количество байтов во входном потоке не кратно 3 (24 битам), последние байты дополняются до полной 24-битовой группы. Для обозначения дополнительных данных используются символы '=', которые добавляются в конец преобразованных данных. Например, при преобразовании 16-бай-

товой последовательности получается пять 3-байтовых групп и остается еще один байт. Этот последний байт дополняется до образования полной 3-байтовой группы. Из этой группы затем образуются два символа из алфавита base 64 (первые 12 бит в которых включают 8 бит фактических данных), за которыми следует пара символов '==', представляющих дополнение. В формате base 64 в конце блока данных допускаются только ноль, один (=) или два (==) символа дополнения.

При использовании формата base 32 двоичные данные объединяются в группы по 40 бит (5 байт). Каждая 40-битная группа делится на восемь 5-битовых компонентов. Далее каждое 5-битное значение представляется с использованием следующего алфавита:

Значение	Символ
0–25	ABCDEFGHIJKLMNOPQRSTUVWXYZ
26–31	2-7

Как и при использовании формата base 64, если в конце входного потока не получается полной 40-битовой группы, она дополняется до 40 бит, а для обозначения дополнительных данных используется символ '='. В выходных данных может присутствовать не более шести символов дополнения ('======'), – когда последняя группа включает в себя только один байт фактических данных.

Для представления данных в формате base 16 используется стандартная шестнадцатеричная форма записи. Каждая 4-битная группа представляется цифрой '0'–'9' или буквой 'A'–'F'. В формате base 16 потребность в дополнении не возникает.

`b64encode(`*`s`* `[,` *`altchars`*`])`

Преобразует строку байтов *s* в формат base 64. В аргументе *altchars* можно передать строку из двух символов, которая определяет альтернативы для символов '+' и '/', обычно используемых для представления данных в формате base 64. Эту особенность можно использовать, когда данные в формате base 64 используются в именах файлов или в адресах URL.

`b64decode(`*`s`* `[,` *`altchars`*`])`

Декодирует строку *s*, содержащую данные в формате base 64, и возвращает строку байтов с декодированными данными. В аргументе *altchars* можно передать строку из двух символов, которая определяет альтернативы для символов '+' и '/', обычно используемых для представления данных в формате base 64. Если входная строка *s* содержит недопустимые символы или некорректное дополнение, функция возбуждает исключение `TypeError`.

`standard_b64encode(`*`s`*`)`

Преобразует строку байтов *s* в стандартный формат base 64.

`standard_b64decode(`*`s`*`)`

Декодирует строку *s*, содержащую данные в стандартном формате base 64.

`urlsafe_b64encode(s)`

Преобразует строку байтов *s* в формат base 64, используя символы '-' и '_' вместо '+' и '/' соответственно. Аналогично вызову `b64encode(s, '-_')`.

`urlsafe_b64decode(s)`

Декодирует строку *s*, содержащую данные в формате base 64, где вместо символов '+' и '/' используются символы '-' и '_' соответственно.

`b32encode(s)`

Преобразует строку байтов *s* в формат base 32.

`b32decode(s [, casefold [, map01]])`

Декодирует строку *s*, содержащую данные в формате **base 32. Если в аргументе** *casefold* передать значение `True`, допустимыми будут считаться алфавитные символы верхнего и нижнего регистров. В противном случае допустимыми будут считаться только символы верхнего регистра (по умолчанию). Аргумент *map01*, если присутствует, определяет символ, в который будет отображаться цифра '1' (например, в символ 'I' или в символ 'L'). Если этот аргумент определен, цифра '0' будет отображаться в символ 'O'. Если входная строка содержит недопустимые символы или некорректное дополнение, функция возбуждает исключение `TypeError`.

`b16encode(s)`

Преобразует строку байтов *s* в формат **base 16 (шестнадцатеричное представление).**

`b16decode(s [,casefold])`

Декодирует строку *s*, содержащую данные в формате **base 16. Если в аргументе** *casefold* передать значение `True`, допустимыми будут считаться алфавитные символы верхнего и нижнего регистров. В противном случае допустимыми будут считаться только символы 'A'—'F' верхнего регистра (по умолчанию). Если входная строка содержит недопустимые символы или какие-либо другие нарушения, функция возбуждает исключение `TypeError`.

Ниже перечислены функции, составляющие прежний интерфейс модуля `base64`, которые могут вам встретиться в существующих программах на языке Python:

`decode(input, output)`

Декодирует данные в формате base 64. В аргументе *input* передается имя файла или объект файла, открытый для чтения. В аргументе *output* передается имя файла или объект файла, открытый для записи в двоичном режиме.

`decodestring(s)`

Декодирует строку *s*, содержащую данные в формате base 64. Возвращает строку с декодированными двоичными данными.

`encode(input, output)`

Преобразует данные в формат base 64. В аргументе *input* передается имя файла или объект файла, открытый для чтения в двоичном режиме. В аргументе *output* передается имя файла или объект файла, открытый для записи.

`encodestring(s)`

Преобразует строку байтов *s* в формат base 64.

Модуль binascii

Модуль `binascii` содержит низкоуровневые функции, используемые для преобразования двоичных данных в различные форматы ASCII и обратно, например форматы base 64, BinHex или UUencoding.

`a2b_uu(s)`

Преобразует текстовую строку *s* с данными в формате uuencode в двоичное представление и возвращает строку байтов. Обычно текстовые строки содержат по 45 байт, за исключением последней, которая может быть меньше. Неполная строка с данными может дополняться пробелами.

`b2a_uu(data)`

Преобразует строку с двоичными данными в текстовую строку в формате uuencode, содержащую только символы ASCII. Длина строки *data* не должна превышать 45 байт, в противном случае будет возбуждено исключение `Error`.

`a2b_base64(string)`

Преобразует строку *string* с данными в формате base 64 в двоичное представление и возвращает строку байтов.

`b2a_base64(data)`

Преобразует строку с двоичными данными в текстовую строку в формате base 64, содержащую только символы ASCII. Длина строки *data* не должна превышать 57 байт, если результат предполагается отправлять по электронной почте (в противном случае лишние данные могут быть отсечены).

`a2b_hex(string)`

Преобразует строку *string* с данными в шестнадцатеричном представлении в двоичные данные. Эта функция может также вызываться как `unhexlify(string)`.

`b2a_hex(data)`

Преобразует строку с двоичными данными в строку с шестнадцатеричным представлением. Эта функция может также вызываться как `hexlify(data)`.

`a2b_hqx(string)`

Преобразует строку с данными в формате BinHex 4 в двоичное представление без выполнения декомпрессии RLE (Run-Length Encoding – кодирование длины повторения).

`rledecode_hqx(data)`

Выполняет RLE-декомпрессию двоичных данных в аргументе *data*. Возвращает распакованные данные, если на входе были получены полные данные, в противном случае возбуждает исключение `Incomplete`.

`rlecode_hqx(`*`data`*`)`

Выполняет BinHex 4 RLE-сжатие данных *data*.

`b2a_hqx(`*`data`*`)`

Преобразует двоичные данные в строку формата BinHex 4, содержащую только символы ASCII. Данные в аргументе *data* уже должны быть сжаты по алгоритму RLE. Кроме того, если это не последний фрагмент данных, длина аргумента *data* должна быть кратна числу 3.

`crc_hqx(`*`data`*`, `*`crc`*`)`

Вычисляет контрольную сумму BinHex 4 CRC строки байтов *data*. В аргументе *crc* передается начальное значение контрольной суммы.

`crc32(`*`data`*` [, `*`crc`*`])`

Вычисляет контрольную сумму CRC-32 строки байтов *data*. В необязательном аргументе *crc* можно передать начальное значение контрольной суммы. Если этот аргумент опущен, аргумент *crc* получает значение по умолчанию `0`.

Модуль csv

Модуль `csv` используется для чтения и записи файлов, содержащих данные, разделенные запятыми (Comma-Separated Values, CSV). Файлы в формате CSV содержат текстовые строки, где каждая строка содержит значения, разделенные символом-разделителем, обычно запятой (`,`) или символом табуляции. Например:

```
Blues,Elwood,"1060 W Addison","Chicago, IL 60613","B263-1655-2187",116,56
```

Разновидности этого формата часто используются в приложениях баз данных и в электронных таблицах. Например, база данных может поддерживать экспорт таблиц в формате CSV, что позволяет другим программам читать содержимое таблиц. Особые сложности начинают возникать, когда фактические данные содержат символ-разделитель. Так, в предыдущем примере одно из полей содержит запятую, и поэтому данное поле было заключено в кавычки. По этой причине для работы с подобными файлами часто бывает недостаточно простых строковых операций, таких как `split(',')`.

`reader(`*`csvfile`*` [, `*`dialect`*` [, `*`**fmtparams`*`])`

Возвращает объект чтения, который воспроизводит значения для каждой строки в исходном файле *csvfile*. В аргументе *csvfile* допускается передавать любой итерируемый объект, который в каждой итерации воспроизводит полную строку текста. Возвращаемый объект является итератором, который в каждой итерации воспроизводит список строк. В аргументе *dialect* можно передать либо строку с названием диалекта, либо объект класса `Dialect`. Назначение аргумента *dialect* состоит в том, чтобы обеспечить учет различий между разными вариантами формата CSV. Единственными встроенными диалектами, которые поддерживаются этим модулем, являются диалекты `'excel'` (по умолчанию) и `'excel-tab'`, однако существует возможность определять собственные диалекты, как будет описано ниже

в этом разделе. В *fmtparams* передается набор именованных аргументов, определяющих различные аспекты диалекта. Ниже перечислены допустимые именованные аргументы:

Именованный аргумент	Описание
delimiter	Символ, используемый для отделения полей друг от друга (по умолчанию используется ‘,’).
doublequote	Логический флаг, который определяет, как будет обрабатываться символ кавычки (quotechar), входящий в значение поля. Если имеет значение True, символ кавычки будет просто продублирован. Если имеет значение False, перед ним будет добавлен экранирующий символ (escapechar). По умолчанию принимает значение True.
escapechar	Символ, используемый как экранирующий символ, когда разделитель появляется внутри поля и при этом параметр quoting имеет значение QUOTE_NONE. По умолчанию принимает значение None.
lineterminator	Последовательность символов, завершающих строку (по умолчанию используется последовательность ‘\r\n’).
quotechar	Символ, используемый в качестве кавычки для заключения поля, содержащего символ-разделитель (по умолчанию используется ‘”’).
skipinitialspace	Если получает значение True, пробельные символы, следующие непосредственно за символом-разделителем, игнорируются (по умолчанию принимает значение False).

writer(*csvfile* [, *dialect* [, ***fmtparam*]])

Возвращает объект записи, который можно использовать для создания файлов формата CSV. В аргументе *csvfile* допускается передавать любой объект, похожий на файл, который поддерживает метод write(). Аргумент *dialect* имеет тот же смысл, что и в функции reader(), и используется для преодоления различий между различными вариантами формата CSV. Аргумент *fmtparams* имеет тот же смысл, что и в функции reader(). Однако в нем допускается указывать еще один именованный аргумент:

Именованный аргумент	Описание
quoting	Управляет заключением выходных данных в кавычки. Может принимать одно из следующих значений: QUOTE_ALL (в кавычки заключаются все поля), QUOTE_MINIMAL (в кавычки заключаются только поля, содержащие символ-разделитель или начинающиеся с символа кавычки), QUOTE_NONNUMERIC (в кавычки заключаются все нечисловые поля) или QUOTE_NONE (поля никогда не заключаются в кавычки). По умолчанию принимает значение QUOTE_MINIMAL.

Экземпляр *w* объекта записи поддерживает следующие методы:

w.writerow(*row*)

Записывает в файл единственную строку данных. Аргумент *row* должен содержать последовательность строковых или числовых значений.

w.writerows(*rows*)

Записывает несколько строк данных. Аргумент *rows* должен быть последовательностью записей, которые передаются методу writerow().

DictReader(*csvfile* [, *fieldnames* [, *restkey* [, *restval* [, *dialect* [, ***fmtparams*]]]]])

Возвращает объект чтения, который действует как обычный объект чтения, но при чтении данных из файла вместо списка строк возвращает словарь объектов. В аргументе *fieldnames* передается список имен полей, которые будут использоваться в качестве ключей возвращаемого словаря. Если этот аргумент опущен, имена полей будут взяты из первой строки исходного файла. В аргументе *restkey* передается имя ключа словаря, который будет использоваться для сохранения лишних данных, например, когда количество полей в записи превышает количество имен полей в аргументе *fieldnames*. Аргумент *restval* определяет значение по умолчанию, которое будет использоваться в качестве значения отсутствующих полей, например, когда количество полей в записи меньше количества имен полей в аргументе *fieldnames*. По умолчанию аргументы *restkey* и *restval* принимают значение None. Аргументы *dialect* и *fmtparams* имеют тот же смысл, что и в функции reader().

DictWriter(*csvfile*, *fieldnames* [, *restval* [, *extrasaction* [, *dialect* [, ***fmtparams*]]]])

Возвращает объект записи, который действует так же, как обычный объект записи, но используется для записи в файл содержимого словарей. Аргумент *fieldnames* определяет имена полей и порядок их следования при записи в файл. Аргумент *restval* определяет значение, которое должно быть записано в файл вместо ключа, указанного в аргументе *fieldnames*, но отсутствующего в словаре. В аргументе *extrasaction* передается строка, которая определяет, что делать, если в словаре имеются ключи, отсутствующие в аргументе *fieldnames*. По умолчанию аргумент *extrasaction* принимает значение 'raise', что соответствует возбуждению исключения ValueError. Можно использовать значение 'ignore', и в этом случае лишние ключи словаря будут игнорироваться. Аргументы *dialect* и *fmtparams* имеют тот же смысл, что и в функции writer().

Экземпляр *w* класса DictWriter поддерживает следующие методы:

w.writerow(*row*)

Записывает в файл единственную запись с данными. Аргумент *row* должен быть словарем, отображающим имена полей в их значения.

w.writerows(*rows*)

Записывает в файл несколько записей данных. Аргумент *rows* должен быть последовательностью записей, которые можно передать методу writerow().

`Sniffer()`

Создает объект класса `Sniffer`, который используется для автоматического определения формата файла CSV.

Экземпляр *s* класса `Sniffer` обладает следующими методами:

s`.sniff(`*sample* `[, ` *delimiters*`])`

Просматривает данные, содержащиеся в *sample*, и возвращает соответствующий объект класса `Dialect`, представляющий формат данных. В аргументе *sample* передается фрагмент файла CSV, содержащий по меньшей мере одну полную запись с данными. В необязательном аргументе *delimiters* можно передать строку с предполагаемыми символами-разделителями.

s`.has_header(`*sample*`)`

Просматривает данные, содержащиеся в *sample*, и возвращает `True`, если первая запись выглядит, как коллекция с заголовками полей.

Диалекты

Многие функции и методы в модуле `csv` имеют специальный параметр, определяющий диалект. Назначение этого параметра состоит в том, чтобы обеспечить возможность настройки под различные разновидности формата CSV (для которого не существует официального «стандарта»), – например, когда значения разделяются запятыми или символами табуляции, когда используются различные соглашения по заключению значений в кавычки и так далее.

Новые диалекты определяются как классы, производные от класса `Dialect` и объявляющие набор атрибутов с теми же именами, что и именованные аргументы функций `reader()` и `writer()` (`delimiter`, `doublequote`, `escapechar`, `lineterminator`, `quotechar`, `quoting`, `skipinitialspace`).

Для управления диалектами используются следующие вспомогательные функции:

`register_dialect(`*name*`, ` *dialect*`)`

Регистрирует новый объект *dialect* класса `Dialect` под именем *name*.

`unregister_dialect(`*name*`)`

Удаляет объект класса `Dialect` с именем *name*.

`get_dialect(`*name*`)`

Возвращает объект класса `Dialect` с именем *name*.

`list_dialects()`

Возвращает список имен зарегистрированных диалектов. В настоящее время существует всего два встроенных диалекта: `'excel'` и `'excel-tab'`.

Пример

```
import csv
# Чтение простого файла CSV
f = open("scmods.csv","r")
```

```
for r in csv.reader(f):
    lastname, firstname, street, city, zip = r
    print("{0} {1} {2} {3} {4}".format(*r))

# Прочитать с помощью объекта класса DictReader
f = open("address.csv")
r = csv.DictReader(f,['lastname','firstname','street','city','zip'])
for a in r:
    print("{firstname} {lastname} {street} {city} {zip}".format(**a))

# Запись простых данных в файл CSV
data = [
     ['Blues','Elwood','1060 W Addison','Chicago','IL','60613' ],
     ['McGurn','Jack','4802 N Broadway','Chicago','IL','60640' ],
]
f = open("address.csv","w")
w = csv.writer(f)
w.writerows(data)
f.close()
```

Пакет email

Пакет `email` предоставляет обширный спектр функций и объектов для представления, анализа и управления сообщениями электронной почты, представленными в формате, соответствующем стандарту MIME.

Рассматривать в этом разделе все особенности пакета `email` не имеет особого смысла, и это вряд ли представляет интерес для большинства читателей. Поэтому в оставшейся части раздела мы сконцентрируемся на изучении двух основных практических проблем: анализе сообщений электронной почты с целью извлечения полезной информации и создании новых сообщений, готовых к отправке средствами модуля `smtplib`.

Анализ сообщений электронной почты

На самом верхнем уровне пакет `email` предоставляет две функции, позволяющие выполнять анализ сообщений электронной почты:

`message_from_file(`*`f`*`)`

Анализирует сообщение электронной почты, читая его из объекта *f*, похожего на файл, открытого для чтения в текстовом режиме. Исходное сообщение должно быть полноценным сообщением электронной почты в формате MIME, включающем все заголовки, текст и вложения. Возвращает экземпляр класса `Message`.

`message_from_string(`*`str`*`)`

Анализирует сообщение электронной почты, читая его из строки *str*. Возвращает экземпляр класса `Message`.

Экземпляр *m* класса `Message`, возвращаемый предыдущими функциями, имитирует словарь и поддерживает следующие операции по поиску данных внутри сообщения:

Операция	Описание
m[*name*]	Возвращает значение заголовка с именем *name*.
m.keys()	Возвращает список имен всех заголовков в сообщении.
m.values()	Возвращает список значений всех заголовков в сообщении.
m.items()	Возвращает список кортежей, содержащих имена и значения заголовков.
m.get(*name* [,*def*])	Возвращает значение заголовка с именем *name*. Аргумент *def* определяет значение по умолчанию, возвращаемое в случае отсутствия искомого заголовка.
len(*m*)	Возвращает количество заголовков в сообщении.
str(*m*)	Преобразует сообщение в строку. Действует так же, как и метод as_string().
name in *m*	Возвращает True, если в сообщении присутствует заголовок с именем *name*.

В дополнение к этим операциям экземпляр *m* обладает следующими методами, которые могут использоваться для извлечения информации:

m.get_all(*name* [, *default*])

Возвращает список всех значений заголовка с именем *name*. Если искомый заголовок отсутствует, возвращает *default*.

m.get_boundary([*default*])

Возвращает параметр boundary, найденный в заголовке 'Content-type' сообщения. Обычно значением этого параметра является строка, такая как '==========0995017162==', которая применяется для отделения различных частей сообщения. Если параметр boundary не найден, возвращает *default*.

m.get_charset()

Возвращает название набора символов, который используется в теле сообщения (например, 'iso-8859-1').

m.get_charsets([*default*])

Возвращает список названий всех наборов символов, присутствующих в сообщении. Для сообщений, состоящих из нескольких частей, в список будет включено название кодировки символов для каждой части. Название набора символов для каждой части извлекается из заголовков 'Content-type', присутствующих в сообщении. Если в какой-либо части название набора символов не указано или отсутствует заголовок 'Content-type', для нее возвращается значение аргумента *default* (по умолчанию принимает значение None).

m.get_content_charset([*default*])

Возвращает название набора символов из первого заголовка 'Content-type' в сообщении. Если такой заголовок отсутствует или в нем не указано название набора символов, возвращается значение аргумента *default*.

m.get_content_maintype()

Возвращает тип основного содержимого (например, 'text' или 'multipart').

m.get_content_subtype()

Возвращает подтип содержимого (например, 'plain' или 'mixed').

m.get_content_type()

Возвращает строку, содержащую описание типа содержимого сообщения (например, 'multipart/mixed' или 'text/plain').

m.get_default_type()

Возвращает тип содержимого по умолчанию (например, 'text/plain' для простых сообщений).

m.get_filename([*default*])

Возвращает значение параметра filename из заголовка 'Content-Disposition', если имеется. Возвращает значение аргумента *default*, если заголовок отсутствует или не содержит параметр filename.

m.get_param(*param* [, *default* [, *header* [, *unquote*]]])

В заголовки сообщений электронной почты часто добавляются дополнительные параметры, например параметры 'charset' и 'format' в заголовке 'Content-Type: text/plain; charset="utf-8"; format=flowed'. Этот метод возвращает значение требуемого параметра заголовка. Аргумент *param* определяет имя параметра, аргумент *default* – значение по умолчанию, возвращаемое в случае отсутствия параметра, аргумент *header* – имя заголовка и аргумент *unquote* определяет, следует ли убирать кавычки, окружающие параметр. Если аргумент *header* опущен, по умолчанию используется заголовок 'Content-type'. Аргумент *unquote* по умолчанию принимает значение True. Возвращаемым значением является либо строка, либо кортеж с тремя элементами (*charset*, *language*, *value*), если параметр представлен в формате, соответствующем соглашениям, изложенным в RFC-2231. В последнем случае в поле *charset* возвращается строка, такая как 'iso-8859-1', в поле *language* – строка с кодом языка, такая как 'en', и в поле *value* – значение параметра.

m.get_params([*default* [, *header* [, *unquote*]]])

Возвращает все параметры заголовка *header* в виде списка. Аргумент *default* определяет значение, возвращаемое в случае отсутствия заголовка. Если аргумент *header* опущен, используется заголовок 'Content-type'. Аргумент *unquote* определяет, следует ли убирать кавычки, окружающие значения параметров (True по умолчанию). Возвращаемый список содержит кортежи вида (*name*, *value*), где в поле *name* содержится имя параметра, а в поле *value* – значение, возвращаемое методом get_param().

m.get_payload([*i* [, *decode*]])

Возвращает содержимое сообщения. Если сообщение имеет простой вид, возвращается строка байтов с телом сообщения. Если сообщение состоит из нескольких частей, возвращается список, включающий все части сообщения. Для сообщений, состоящих из нескольких частей, необязатель-

ный аргумент *i* задает индекс в этом списке. Если этот аргумент указан, возвращается только указанный компонент сообщения. Если в аргументе *decode* передается значение True, содержимое декодируется в соответствии с параметрами заголовка 'Content-Transfer-Encoding', который может присутствовать в сообщении (например, 'quoted-printable', 'base64' и так далее). Чтобы декодировать содержимое простого сообщения, состоящего из одной части, следует в аргументе *i* передать значение None, а в аргументе *decode* — значение True или определить аргумент *decode* как именованный аргумент. Следует подчеркнуть, что содержимое возвращается как строка байтов с содержимым. Если содержимым является текст в кодировке UTF-8 или какой-либо другой кодировке, необходимо результат этого метода передать методу decode(), чтобы преобразовать его.

m.get_unixfrom()

Возвращает строку 'From ...' в стиле UNIX, если имеется.

m.is_multipart()

Возвращает True, если объект *m* представляет сообщение, состоящее из нескольких частей.

m.walk()

Создает генератор, который выполняет итерации по всем частям сообщения, каждая из которых также будет представлена экземпляром класса Message. Итерации выполняются сначала в глубину каждой части сообщения, а потом происходит переход к следующей части. Обычно эта функция используется для обработки всех компонентов сложных составных сообщений.

Наконец, экземпляры класса Message обладают несколькими атрибутами, имеющими отношение к низкоуровневым параметрам процесса анализа.

m.preamble

Любой текст, присутствующий в сообщении, состоящем из нескольких частей, между пустой строкой, обозначающей конец блока заголовков, и первым вхождением строки границы, разделяющей части сообщения и отмечающей начало первой части сообщения.

m.epilogue

Любой текст, присутствующий в сообщении, состоящем из нескольких частей, следующий за последним вхождением строки границы в конце сообщения.

m.defects

Список всех дефектов, обнаруженных при анализе сообщения. Дополнительные подробности можно найти в документации к модулю email.errors.

Следующий пример иллюстрирует, как можно использовать класс Message для анализа сообщения электронной почты. Данный программный код читает сообщение электронной почты, выводит краткую информацию о заголовках, выводит текстовые фрагменты сообщения и сохраняет вложения.

```
import email
import sys

f = open(sys.argv[1],"r")       # Открыть файл с сообщением
m = email.message_from_file(f)  # Выполнить анализ

# Вывести информацию об отправителе/получателе
print("From    : %s" % m["from"])
print("To      : %s" % m["to"])
print("Subject : %s" % m["subject"])
print("")

if not m.is_multipart():
    # Простое сообщение. Просто вывести текст сообщения
    payload = m.get_payload(decode=True)
    charset = m.get_content_charset('iso-8859-1')
    print(payload.decode(charset))
else:
    # Сообщение состоит из нескольких частей. Обойти все части и
    #     1. Вывести фрагменты типа text/plain
    #     2. Сохранить вложения
    for s in m.walk():
        filename = s.get_filename()
        if filename:
            print("Сохранение вложения: %s" % filename)
            data = s.get_payload(decode=True)
            open(filename,"wb").write(data)
        else:
            if s.get_content_type() == 'text/plain':
                payload = s.get_payload(decode=True)
                charset = s.get_content_charset('iso-8859-1')
                print(payload.decode(charset))
```

В этом примере важно отметить, что операции, извлекающие содержимое из сообщения, всегда возвращают строки байтов. Если содержимое является обычным текстом, его следует преобразовать в соответствии с требуемой кодировкой. Такое преобразование в данном примере выполняется с помощью вызовов методов `m.get_content_charset()` и `payload.decode()`.

Составление сообщений электронной почты

Чтобы составить новое сообщение электронной почты, необходимо либо создать новый экземпляр класса `Message`, который определен в модуле `email.message`, либо использовать экземпляр класса `Message`, созданный ранее при анализе сообщения (смотрите предыдущий раздел).

`Message()`

Создает новое пустое сообщение.

Экземпляр *m* класса `Message` поддерживает следующие методы, позволяющие наполнять сообщение содержимым, добавлять заголовки и другую информацию.

m.add_header(*name*, *value*, ***params*)

Добавляет новый заголовок сообщения. В аргументе *name* передается имя заголовка, в аргументе *value* – значение заголовка и в *params* – набор именованных аргументов, представляющих дополнительные необязательные параметры. Например, вызов add_header('Foo','Bar',spam='major') добавит в сообщение строку заголовка 'Foo: Bar; spam="major"'.

m.as_string([*unixfrom*])

Преобразует все сообщение в строку. В аргументе *unixfrom* передается логический флаг. Если он имеет значение True, первой строкой сообщения будет строка 'From ...' в стиле UNIX. По умолчанию аргумент *unixfrom* принимает значение False.

m.attach(payload)

Добавляет вложение в сообщение, состоящее из нескольких частей. В аргументе *payload* должен передаваться другой объект класса Message (например, email.mime.text.MIMEText). За кулисами объект *payload* добавляется в конец списка, хранящий различные части сообщения. Если сообщение состоит из единственной части, следует использовать метод set_payload(), который записывает простую строку в тело сообщения.

m.del_param(*param* [, *header* [, *requote*]])

Удаляет параметр *param* из заголовка *header*. Например, если в сообщении присутствует заголовок 'Foo: Bar; spam="major"', вызов del_param('spam','Foo') удалит фрагмент 'spam="major"' из заголовка. Если в аргументе *requote* передается значение True (по умолчанию), все остающиеся значения будут заключены в кавычки при перезаписи заголовка. Если опустить аргумент *header*, действие будет выполнено над заголовком 'Content-type'.

m.replace_header(*name*, *value*)

Замещает первое вхождение заголовка с именем *name* значением *value*. Если искомый заголовок отсутствует, возбуждает исключение KeyError.

m.set_boundary(*boundary*)

Устанавливает параметр boundary сообщения в значение *boundary*. Эта строка добавляется в заголовок 'Content-type' сообщения в виде параметра boundary. Если в сообщении отсутствует заголовок 'Content-type', возбуждает исключение HeaderParseError.

m.set_charset(*charset*)

Устанавливает для сообщения кодировку символов по умолчанию. Аргумент *charset* может быть строкой, такой как 'iso-8859-1' или 'euc-jp'. При установке кодировки в заголовок 'Content-type' сообщения обычно добавляется параметр (например, 'Content-type: text/html; charset="iso-8859-1"').

m.set_default_type(*ctype*)

Устанавливает для сообщения тип содержимого по умолчанию *ctype*. В аргументе *ctype* передается строка, содержащая название типа MIME, такая как 'text/plain' или 'message/rfc822'. Этот тип не сохраняется в заголовке 'Content-type' сообщения.

m.set_param(*param*, *value* [, *header* [, *requote* [, *charset* [, *language*]]]])

Устанавливает значение параметра заголовка. В аргументе *param* передается имя параметра, а в аргументе *value* – его значение. Аргумент *header* определяет имя заголовка, по умолчанию подразумевается заголовок 'Content-type'. Аргумент *requote* определяет необходимость заключения в кавычки всех значений, присутствующих в заголовке, после добавления параметра. По умолчанию этот аргумент принимает значение True. Аргументы *charset* и *language* определяют дополнительную информацию о кодировке символов и языке. Если эти аргументы заданы, параметры сохраняются в формате RFC-2231. Этот метод добавляет текст с определением параметра вида param*="'iso-8859-1'en-us'some%20value".

m.set_payload(*payload* [, *charset*])

Сохраняет значение аргумента *payload* как содержимое сообщения. Для простых сообщений в аргументе *payload* можно передать строку байтов с телом сообщения. Для сообщений, состоящих из нескольких частей, в аргументе *payload* передается список объектов класса Message. Необязательный аргумент *charset* определяет кодировку символов, которая была использована при создании текста (смотрите описание метода set_charset()).

m.set_type(*type* [, *header* [, *requote*]])

Устанавливает тип содержимого, который будет записан в заголовок 'Content-type'. В аргументе *type* передается строка, такая как 'text/plain' или 'multipart/mixed', определяющая тип. Аргумент *header* определяет альтернативный заголовок, отличный от заголовка по умолчанию 'Content-type'. Аргумент *requote* определяет необходимость заключения в кавычки всех значений, уже присутствующих в заголовке, после добавления параметра. По умолчанию принимает значение True.

m.set_unixfrom(*unixfrom*)

Устанавливает текст для строки 'From ...' в стиле UNIX. В аргументе *unixfrom* должна передаваться строка, содержащая полный текст, включая префикс 'From'. Этот текст выводится, только если в вызове метода *m*.as_string() аргумент unixfrom получает значение True.

Вместо того, чтобы создавать низкоуровневые объекты класса Message и каждый раз конструировать сообщения с нуля, можно использовать уже готовые объекты сообщений, сконструированные для различных типов содержимого. Эти объекты особенно удобно использовать при создании сообщений, состоящих из нескольких частей. Например, можно создать новое сообщение и присоединить к нему различные части, воспользовавшись методом attach() объекта класса Message. Каждый из этих объектов определяется в собственном, отдельном модуле, название которого указывается в описании.

MIMEApplication(*data* [, *subtype* [, *encoder* [, ***params*]]])

Объявляется в модуле email.mime.application. Создает сообщение, содержащее данные приложения. В аргументе *data* передается строка байтов с двоичными данными. Аргумент *subtype* определяет подтип данных и по

умолчанию принимает значение 'octet-stream'. В необязательном аргументе *encoder* передается функция кодирования из подпакета email.encoders. По умолчанию данные преобразуются в формат base 64. Аргумент *params* представляет дополнительные именованные аргументы и значения, которые должны быть добавлены в заголовок 'Content-type' сообщения.

`MIMEAudio(`*data* `[, ` *subtype* `[, ` *encoder* `[, **`*params*`]]])`

Объявляется в модуле email.mime.audio. Создает сообщение, содержащее аудиоданные. В аргументе *data* передается строка байтов с двоичными аудиоданными. Аргумент *subtype* определяет тип аудиоданных и может быть строкой, такой как 'mpeg' или 'wav'. Если аргумент *subtype* отсутствует, предположительный тип аудиоданных будет определен автоматически, с помощью модуля sndhdr. Аргументы *encoder* и *params* имеют тот же смысл, что и в функции MIMEApplication().

`MIMEImage(`*data* `[, ` *subtype* `[, ` *encoder* `[, **`*params*`]]])`

Объявляется в модуле email.mime.image. Создает сообщение, содержащее изображение. В аргументе *data* передается строка байтов с двоичным представлением изображения. Аргумент *subtype* определяет тип изображения и может быть строкой, такой как 'jpg' или 'png'. Если аргумент *subtype* отсутствует, предположительный тип изображения будет определен автоматически, с помощью функции из модуля imghdr. Аргументы *encoder* и *params* имеют тот же смысл, что и в функции MIMEApplication().

`MIMEMessage(`*msg* `[, ` *subtype*`])`

Объявляется в модуле email.mime.message. Создает новое сообщение MIME, состоящее из одной части. В аргументе *msg* передается объект сообщения с начальным содержимым. Аргумент *subtype* определяет тип сообщения и по умолчанию принимает значение 'rfc822'.

`MIMEMultipart([`*subtype* `[, ` *boundary* `[, ` *subparts* `[, **`*params*`]]]])`

Объявляется в модуле email.mime.multipart. Создает новое сообщение MIME, состоящее из нескольких частей. Аргумент *subtype* определяет необязательный подтип, который будет добавлен в заголовок 'Content-type: multipart/subtype'. По умолчанию принимает значение 'mixed'. В аргументе *boundary* передается строка, которая будет использоваться в качестве границы, отделяющей части сообщения. Если этот аргумент опущен или в нем передается значение None, подходящая строка границы будет подобрана автоматически. В аргументе *subparts* передается последовательность объектов класса Message, составляющих содержимое сообщения. Аргумент *params* представляет дополнительные именованные аргументы и значения, которые должны быть добавлены в заголовок 'Content-type' сообщения. После создания сообщения, состоящего из нескольких частей, добавлять в него новые части можно будет с помощью метода Message.attach().

`MIMEText(`*data* `[, ` *subtype* `[, ` *charset*`]])`

Объявляется в модуле email.mime.text. Создает сообщение с текстовыми данными. В аргументе *data* передается строка с текстом сообщения. Аргумент *subtype* определяет тип текста и может быть строкой, такой как 'plain'

(по умолчанию) или 'html'. Аргумент *charset* определяет кодировку символов и по умолчанию принимает значение 'us-ascii'. Кодирование сообщения выполняется в зависимости от его содержимого.

Следующий пример демонстрирует, как можно составить сообщение и отправить его по электронной почте, используя классы, представленные в этом разделе:

```
import smtplib
from email.mime.text import MIMEText
from email.mime.multipart import MIMEMultipart
from email.mime.audio import MIMEAudio

sender   = "jon@nogodiggydie.net"
receiver = "dave@dabeaz.com"
subject  = "Faders up!"
body     = "I never should have moved out of Texas. -J.\n"
audio    = "TexasFuneral.mp3"

m = MIMEMultipart()
m["to"]      = receiver
m["from"]    = sender
m["subject"] = subject

m.attach(MIMEText(body))
apart = MIMEAudio(open(audio,"rb").read(),"mpeg")
apart.add_header("Content-Disposition","attachment",filename=audio)
m.attach(apart)

# Отправить сообщение по электронной почте
s = smtplib.SMTP()
s.connect()
s.sendmail(sender, [receiver],m.as_string())
s.close()
```

Примечания

- Многие из дополнительных параметров и настроек здесь не рассматривались. Тем, кому потребуется воспользоваться дополнительными возможностями этого пакета, рекомендуется обращаться к электронной документации.
- В процессе развития пакета email было выпущено по меньшей мере четыре различные версии, в которых программный интерфейс достаточно существенно изменялся (то есть модули пакета переименовывались, определения классов переносились в другие модули и так далее). В этом разделе был описан интерфейс версии 4.0, которая используется в Python 2.6 и Python 3.0. Если вам доведется сопровождать устаревший программный код, вы также сможете применять изложенные здесь концепции, но вам придется изменить местоположение классов и имена модулей.

Модуль hashlib

Модуль `hashlib` реализует различные алгоритмы вычисления контрольных сумм сообщений, такие как **MD5 и SHA1. Чтобы вычислить значение контрольной суммы**, сначала требуется вызвать одну из следующих функций, имена которых соответствуют названиям алгоритмов:

Функция	Описание
`md5()`	Контрольная сумма MD5 (128 бит)
`sha1()`	Контрольная сумма SHA1 (160 бит)
`sha224()`	Контрольная сумма SHA224 (224 бита)
`sha256()`	Контрольная сумма SHA256 (256 бит)
`sha384()`	Контрольная сумма SHA384 (384 бита)
`sha512()`	Контрольная сумма SHA512 (512 бит)

Экземпляр *d* объекта контрольной суммы, возвращаемый любой из этих функций, имеет следующий интерфейс:

Метод или атрибут	Описание
d.`update(`*data*`)`	Обновляет контрольную сумму с учетом добавления новых данных. Аргумент *data* должен быть строкой байтов. Последовательность нескольких вызовов равноценна одному вызову для всего объема данных.
d.`digest()`	Возвращает контрольную сумму в виде строки двоичных байтов.
d.`hexdigest()`	Возвращает текстовую строку со значением контрольной суммы, представленной в виде последовательности шестнадцатеричных цифр.
d.`copy()`	Возвращает копию объекта контрольной суммы. Копия наследует значения внутренних переменных оригинала.
d.`digest_size`	Размер контрольной суммы в байтах.
d.`block_size`	Внутренний размер блока в байтах, используемый алгоритмом хеширования.

Кроме того, модулем предоставляется альтернативный интерфейс создания объектов контрольных сумм:

`new(`*hashname*`)`

Создает новый объект контрольной суммы. В аргументе *hashname* передается строка с названием алгоритма хеширования, такая как `'md5'` или `'sha256'`. В качестве имени алгоритма можно использовать имена алгоритмов, представленные выше, а также имена алгоритмов, объявленные в библиотеке OpenSSL (зависит от параметров сборки интерпретатора).

Модуль hmac

Модуль hmac обеспечивает поддержку механизма HMAC (Keyed-Hashing for Message Authentication – хеш-код аутентификации сообщений), который описывается в RFC-2104. HMAC – это механизм, используемый для аутентификации сообщений, основанный на криптографических функциях вычисления контрольных сумм, таких как MD5 и SHA-1.

`new(`*`key`* `[, `*`msg`* `[, `*`digest`*`]])`

Создает новый объект HMAC, где *key* – это строка байтов с начальным значением хеша, *msg* – начальные данные для обработки и *digest* – конструктор объекта контрольной суммы, который будет использоваться при вычислении хеш-кода. По умолчанию в качестве конструктора используется функция hashlib.md5(). Обычно начальное значение ключа определяется как случайное число, полученное с применением генератора случайных чисел, обладающего улучшенной криптографической стойкостью.

Объект *h* класса HMAC обладает следующими методами:

h`.update(`*`msg`*`)`

Добавляет строку *msg* в объект класса HMAC.

h`.digest()`

Возвращает контрольную сумму всех данных, обработанных к настоящему моменту, в виде строки байтов, содержащую значение контрольной суммы. Длина строки с результатом зависит от используемого алгоритма хеширования. Для MD5 это будет строка из 16 символов; для SHA-1 – из 20 символов.

h`.hexdigest()`

Возвращает контрольную сумму в виде строки из шестнадцатеричных цифр.

h`.copy()`

Создает копию объекта HMAC.

Пример

Основное назначение модуля hmac заключается в том, чтобы обеспечить аутентификацию отправителя сообщения. Для этого в аргументе *key* функции new() передается строка байтов с секретным ключом, известным обеим сторонам – отправителю и получателю сообщения. Посылая сообщение, отправитель создает новый объект HMAC с заданным ключом, обновляет его содержимым сообщения и затем отправляет сообщение вместе с получившейся контрольной суммой HMAC. Принимающая сторона проверяет сообщение, вычисляет собственную контрольную сумму HMAC (используя тот же ключ и содержимое сообщения) и сравнивает полученный результат со значением контрольной суммы, принятым вместе с сообщением. Например:

```
import hmac
secret_key = b"peekaboo" # Строка байтов, известная только мне. Обычно
                         # предпочтительнее использовать строку случайных
```

```
                        # байтов, сгенерированных с помощью os.urandom() или
                        # или подобной ей функции.

data = b"Hello World"   # Текст сообщения

# Отправка сообщения
# out - это сокет или какой-либо другой канал ввода-вывода,
# используемый для отправки данных.
h = hmac.new(secret_key)
h.update(data)
out.send(data)          # Отправка данных
out.send(h.digest())    # Отправка контрольной суммы

# Прием сообщения
# in - это сокет или какой-либо другой канал ввода-вывода,
# используемый для приема данных.
h = hmac.new(secret_key)
data = in.receive()     # Прием данных сообщения
h.update(data)
digest = in.receive()   # Прием контрольной суммы, отправленной отправителем
if digest != h.digest():
    raise AuthenticationError('Message not authenticated')
```

Модуль HTMLParser

В Python 3 этот модуль называется `html.parser`. Модуль `HTMLParser` определяет класс `HTMLParser`, который может использоваться для синтаксического анализа документов HTML и XHTML. Чтобы воспользоваться этим модулем, в программе необходимо объявить собственный класс, производный от класса `HTMLParser`, и переопределить необходимые методы.

`HTMLParser()`

Базовый класс, используемый для создания парсеров разметки HTML. Не имеет входных аргументов.

Экземпляр *h* класса `HTMLParser` обладает следующими методами:

h`.close()`

Закрывает парсер и принудительно запускает анализ всех необработанных данных. Этот метод вызывается после того, как все данные HTML будут переданы парсеру.

h`.feed(`*data*`)`

Передает парсеру новые данные. Эти данные обрабатываются немедленно. Однако если данные неполные (например, завершаются незакрытым элементом HTML), незаконченный фрагмент будет сохранен в буфере и обработан при следующем вызове метода `feed()`, когда будут получены недостающие данные.

h`.getpos()`

Возвращает номер текущей строки и номер текущего символа в строке в виде кортежа (*line*, *offset*).

h.get_starttag_text()

Возвращает текст, соответствующий самому последнему открытому начальному тегу.

h.handle_charref(*name*)

Этот метод-обработчик вызывается всякий раз, когда встречается ссылка на символ, такая как '&#ref;'. В аргументе *name* передается строка со значением ссылки. Например, при встрече ссылки 'å' в аргументе *name* будет передана строка '229'.

h.handle_comment(*data*)

Этот метод-обработчик вызывается всякий раз, когда встречается комментарий. В аргументе *data* передается строка с текстом комментария. Например, при встрече комментария '<!--comment-->' в аргументе *data* будет передана строка 'comment'.

h.handle_data(*data*)

Этот метод-обработчик вызывается для обработки данных, находящихся между открывающим и закрывающим тегами. В аргументе *data* передается строка с текстом.

h.handle_decl(*decl*)

Этот метод-обработчик вызывается для обработки объявлений, таких как '<!DOCTYPE HTML ...>'. В аргументе *decl* передается строка с текстом объявления, за исключением начальной последовательности символов '<!' и завершающего символа '>'.

h.handle_endtag(*tag*)

Этот метод-обработчик вызывается, когда встречается закрывающий тег. В аргументе *tag* передается имя тега, в котором все символы преобразованы в нижний регистр. Например, при встрече закрывающего тега '</BODY>' в аргументе *tag* будет передана строка 'body'.

h.handle_entityref(*name*)

Этот метод-обработчик вызывается для обработки мнемоник, таких как '&*name*;'. В аргументе *name* передается строка с именем мнемоники. Например, при встрече мнемоники '<' в аргументе *name* будет передана строка 'lt'.

h.handle_pi(*data*)

Этот метод-обработчик вызывается для обработки инструкций, таких как '<?*processing instruction*>'. В аргументе *data* передается строка с текстом обрабатываемой инструкции, за исключением начальной последовательности символов '<?' и завершающего символа '>'. Когда встречаются инструкции на языке разметки XHTML, имеющие вид '<?...?>', последний символ '?' включается в строку *data*.

h.handle_startendtag(*tag*, *attrs*)

Этот метод-обработчик вызывается для обработки пустых тегов разметки XHTML, таких как '<tag name="value"…/>'. В аргументе *tag* передается стро-

ка с именем тега. В аргументе *attrs* передается информация об атрибутах в виде списка кортежей вида (*name*, *value*), где *name* – имя атрибута, содержащее только символы нижнего регистра, а *value* – значение атрибута. При извлечении значений кавычки и мнемоники замещаются. Например, при встрече тега '<a href="http://www.foo.com"/>' в аргументе *tag* будет передана строка 'a', а в аргументе *attrs* – список [('href','http://www.foo.com')]. Если этот метод не переопределен в производном классе, реализация по умолчанию просто вызывает методы handle_starttag() и handle_endtag().

h.handle_starttag(*tag*, *attrs*)

Этот метод-обработчик вызывается, когда встречается открывающий тег вида '<tag name="value" ...>'. Аргументы *tag* и *attrs* имеют тот же смысл, что и в методе handle_startendtag().

h.reset()

Сбрасывает парсер в исходное состояние, уничтожает все необработанные данные.

Модуль HTMLParser определяет следующее исключение:

HTMLParserError

Это исключение возбуждается в результате появления ошибок при синтаксическом анализе. Экземпляр исключения HTMLParserError обладает тремя атрибутами. Атрибут msg содержит сообщение с описанием ошибки, атрибут lineno – номер строки, где возникла ошибка, и атрибут offset – номер символа в строке.

Пример

Следующий пример получает страницу HTML, используя для этого пакет urllib, и выводит все ссылки вида '<a href="...">':

```
# printlinks.py
try:
    from HTMLParser import HTMLParser
    from urllib2 import urlopen
except ImportError:
    from html.parser import HTMLParser
    from urllib.request import urlopen
import sys

class PrintLinks(HTMLParser):
    def handle_starttag(self,tag,attrs):
        if tag == 'a':
            for name,value in attrs:
                if name == 'href': print(value)

p    = PrintLinks()
u    = urlopen(sys.argv[1])
data = u.read()
charset = u.info().getparam('charset')     # Python 2
#charset = u.info().get_content_charset()  # Python 3
p.feed(data.decode(charset))
p.close()
```

В этом примере важно отметить, что разметка HTML, получаемая с помощью пакета `urllib`, возвращается в виде строки байтов. Чтобы выполнить синтаксический анализ, эту строку байтов следует декодировать в текст, в соответствии с кодировкой символов, используемой в документе. Пример демонстрирует, как определить кодировку в Python 2 и в Python 3.

Примечание

Модуль `HTMLParser` имеет достаточно ограниченные возможности синтаксического анализа. Слишком сложная разметка HTML или разметка с ошибками оформления может привести к ошибке в парсере и невозможности полного синтаксического анализа. Кроме того, пользователи считают, что этот модуль слишком низкоуровневый и потому недостаточно удобный. Если вам необходим инструмент, с помощью которого можно было бы извлекать данные из страниц HTML, обратите внимание на пакет Beautiful Soup (*http://pypi.python.org/pypi/BeautifulSoup*).

Модуль json

Модуль `json` используется для преобразования объектов в последовательную форму представления и обратно с использованием нотации объектов JavaScript (JavaScript Object Notation, JSON). Более подробную информацию о формате JSON можно найти на странице *http://json.org*.[1] В действительности этот формат является всего лишь подмножеством синтаксиса языка JavaScript. Но в данном случае он не соответствует синтаксису языка Python, используемому для представления списков и словарей. Например, массив в формате JSON записывается как [*value1, value2, ...*], а объект – как {*name*:*value, name*:*value, ...* }.

Ниже перечислены типы данных в формате JSON и соответствующие им типы данных в языке Python. Типы данных в языке Python, указанные в круглых скобках, допускается использовать при преобразовании в формат JSON, но они не возвращаются при обратном преобразовании (возвращается объект первого типа в списке).

Тип JSON	Тип Python
object	dict
array	list (tuple)
string	unicode (str, bytes)
number	int, float
true	True
false	False
null	None

[1] На русском языке: *http://www.json.org/json-ru.html. – Прим. перев.*

При работе со строковыми данными следует исходить из предположения, что все они представлены строками Юникода. Если потребуется преобразовать строку байтов, ее следует декодировать в строку Юникода, используя кодировку `'utf-8'` по умолчанию (впрочем, этим преобразованием вполне можно управлять). При преобразовании строк из формата JSON всегда возвращаются строки Юникода.

Для преобразования в формат JSON и обратно используются следующие функции:

`dump(`*`obj`*`, `*`f`*`, **`*`opts`*`)`

Преобразует объект *obj* в формат JSON и сохраняет его в объекте *f*, похожем на файл. В *opts* передается коллекция именованных аргументов, управляющих процессом преобразования:

Именованный аргумент	Описание
`skipkeys`	Логический флаг, который определяет, что следует делать, когда в качестве ключей (не значений) словаря используются значения не простых типов, таких как строки или числа. Если имеет значение `True`, такие ключи будут пропускаться. Если имеет значение `False` (по умолчанию), будет возбуждаться исключение `TypeError`.
`ensure_ascii`	Логический флаг, который определяет, могут ли строки Юникода записываться в файл *f*. По умолчанию принимает значение `False`. Этот флаг следует устанавливать в значение `True`, только если аргумент *f* представляет объект файла, который корректно работает с Юникодом, например объект файла, созданный с помощью модуля `codecs` или открытый с определенными настройками кодировки.
`check_circular`	Логический флаг, который определяет, следует ли проверять наличие циклических ссылок в контейнерах. По умолчанию получает значение `True`. Если этот аргумент установлен в значение `False` и будет встречена циклическая ссылка, функция возбудит исключение `OverflowError`.
`allow_nan`	Логический флаг, который определяет, будут ли преобразовываться числа с плавающей точкой, со значениями, находящимися за пределами представления (например, `NaN`, `inf`, `-inf`). По умолчанию принимает значение `True`.
`cls`	Подкласс класса `JSONEncoder`, используемый для преобразования. Этот аргумент следует указывать, если был создан собственный класс, производный от класса `JSONEncoder`, выполняющий преобразование. Любые дополнительные именованные аргументы, которые передаются функции `dump()`, передаются конструктору этого класса.
`indent`	Неотрицательное целое число, которое определяет величину отступа при форматировании элементов массивов и атрибутов объектов. Установка этого аргумента обеспечивает более удобочитаемое представление. По умолчанию принимает значение `None`, что соответствует наиболее компактному представлению.

Именованный аргумент	Описание
separators	Кортеж вида (*item_separator*, *dict_separator*), где поле *item_separator* является строкой с символами-разделителями, которые используются для отделения элементов массива, а поле *dict_separator* – строкой с символами-разделителями, которые используются для ключей и значений в словарях. По умолчанию принимают значение (', ', ': ').
encoding	Кодировка, используемая для преобразования строк Юникода. По умолчанию принимает значение 'utf-8'.
default	Функция, используемая для сериализации объектов, которые не относятся ни к одному из основных поддерживаемых типов. Эта функция либо должна возвращать значение, пригодное для сериализации (например, строку), либо возбуждать исключение TypeError. По умолчанию для неподдерживаемых типов возбуждается исключение TypeError.

`dumps(obj, **opts)`

То же, что и `dump()`, за исключением того, что результат преобразования возвращается в виде строки.

`load(f, **opts)`

Принимает объект *f*, похожий на файл, с содержимым в формате JSON и возвращает результат преобразования в объект языка Python. В *opts* передается набор именованных аргументов, управляющих процессом преобразования, которые описываются ниже. Важно помнить, что эта функция вызывает метод *f*.`read()`, возвращающий содержимое файла *f* целиком. Поэтому данной функции не следует передавать такие потоки ввода, как сокеты, где данные в формате JSON могут составлять лишь часть объемного или непрерывного потока данных.

Именованный аргумент	Описание
encoding	Кодировка, используемая при интерпретации всех строковых значений. По умолчанию используется кодировка 'utf-8'.
strict	Логический флаг, который определяет, могут ли появляться в строках JSON литералы (неэкранированные) символов перевода строки. По умолчанию получает значение True, то есть при появлении таких литералов будет возбуждаться исключение.
cls	Подкласс класса JSONDecoder, используемый для преобразования. Этот аргумент следует указывать, если был создан собственный класс, производный от класса JSONDecoder, выполняющий преобразование. Любые дополнительные именованные аргументы, которые передаются функции load(), передаются конструктору этого класса.

(продолжение)

Именованный аргумент	Описание
object_hook	Функция, которая применяется к результату преобразования каждого объекта JSON. По умолчанию используется встроенная функция dict().
parse_float	Функция, которая используется для декодирования значений с плавающей точкой в формате JSON. По умолчанию используется встроенная функция float().
parse_int	Функция, которая используется для декодирования целочисленных значений в формате JSON. По умолчанию используется встроенная функция int().
parse_constant	Функция, которая используется для декодирования констант JSON, таких как 'NaN', 'true', 'false' и других.

`loads(`*s*`, **`*opts*`)`

То же, что и `load()`, за исключением того, что данные в формате JSON передаются в виде строки *s*.

Хотя эти функции называются точно так же, как функции в модулях `pickle` и `marshal`, и они тоже используются для сериализации данных, тем не менее используются они совершенно иначе. В частности, функция `dump()` не должна использоваться для записи более одного объекта JSON в один и тот же файл. Точно так же функция `load()` не в состоянии прочитать более одного объекта JSON из одного и того же файла (если во входном файле будет храниться более одного объекта, в процессе его чтения возникнет ошибка). Объекты в формате JSON следует воспринимать так же, как документы HTML или XML. Ведь обычно ни у кого не возникает желания взять два совершенно разных документа XML и сохранить их в одном и том же файле.

Если потребуется изменить порядок преобразования в формат JSON или обратно, это можно сделать, определив свой класс, производный от одного из следующих базовых классов:

`JSONDecoder(**`*opts*`)`

Класс, реализующий преобразование данных из формата JSON. В *opts* передается набор именованных аргументов, управляющих процессом преобразования, назначение которых приводится в описании функции `load()`.

Экземпляр *d* класса `JSONDecoder` обладает следующими двумя методами:

d`.decode(`*s*`)`

Возвращает представление объекта JSON на языке Python. В аргументе *s* передается строка с объектом JSON.

d`.raw_decode(`*s*`)`

Возвращает кортеж (*pyobj*, *index*), где в поле *pyobj* содержится представление объекта JSON на языке Python, а в поле *index* – позиция в строке *s*, где заканчивается объект JSON. Это обстоятельство может использоваться

при работе с потоками ввода, содержащими дополнительные данные, следующие за объектом JSON.

`JSONEncoder(**opts)`

Класс, реализующий преобразование объекта на языке Python в формат JSON. В *opts* передается набор именованных аргументов, управляющих процессом преобразования, назначение которых приводится в описании функции `dump()`.

Экземпляр *e* класса `JSONEncoder` обладает следующими методами:

`e.default(obj)`

Этот метод вызывается, когда объект *obj* на языке Python не может быть преобразован, если следовать обычным правилам преобразования. Должен возвращать результат, который можно будет преобразовать в формат JSON (например, строку, список или словарь).

`e.encode(obj)`

Преобразует объект *obj* на языке Python в формат JSON.

`e.iterencode(obj)`

Создает итератор, который воспроизводит строки с представлением объекта *obj* на языке Python в формате JSON по мере их составления. Процесс создания строки JSON имеет рекурсивную природу. Например, в процессе преобразования выполняются итерации по ключам словаря и обход других словарей и списков, найденных по пути. Используя этот метод, можно организовать обработку результата по частям, в противоположность способу, когда полный результат возвращается в виде одной большой строки.

При объявлении подклассов, производных от классов `JSONDecoder` и `JSONEncoder`, где переопределяется метод `__init__()`, следует помнить о необходимости обработки именованных аргументов. Ниже показано, как это можно реализовать:

```
class MyJSONDecoder(JSONDecoder):
    def __init__(self, **kwargs):
        # Извлечь мои собственные аргументы
        foo = kwargs.pop('foo',None)
        bar = kwargs.pop('bar',None)
        # Вызвать родительский метод, передав остальные именованные аргументы
        JSONDecoder.__init__(self,**kwargs)
```

Модуль mimetypes

Модуль `mimetypes` используется для определения типа MIME файла, исходя из расширения в имени файла. Он также позволяет получить стандартное расширение имени файла для того или иного типа MIME. Типы MIME состоят из пар тип/подтип, например: '`text/html`', '`image/png`' или '`audio/mpeg`'.

`guess_type(filename [, strict])`

Определяет тип MIME файла, опираясь на имя файла или URL. Возвращает кортеж (*type, encoding*), где поле *type* содержит строку вида "*тип/подтип*",

а поле *encoding* – название формата передаваемых данных (например, compress или gzip). Возвращает (None, None), если тип определить не удалось. Если в аргументе *strict* передается значение True (по умолчанию), распознаваться будут только официальные типы MIME, зарегистрированные в IANA (*http://www.iana.org/assignments/media-types*). В противном случае будут распознаваться некоторые распространенные, но официально не зарегистрированные типы MIME.

`guess_extension(`*`type`* `[, ` *`strict`*`])`

Определяет расширение имени файла по указанному типу MIME. Возвращает строку с расширением, включая начальный символ точки (.). Для неизвестных типов возвращается значение None. Если в аргументе *strict* передается значение True (по умолчанию), распознаваться будут только официальные типы MIME

`guess_all_extensions(`*`type`* `[, ` *`strict`*`])`

То же, что и guess_extension(), но возвращает список возможных расширений имен файлов.

`init([`*`files`*`])`

Инициализирует модуль. В аргументе *files* передается последовательность имен файлов с информацией о типах. Эти файлы содержат строки, отображающие типы MIME в списки допустимых расширений имен файлов, например:

```
image/jpeg: jpe jpeg jpg
text/html: htm html
...
```

`read_mime_types(`*`filename`*`)`

Загружает отображение из файла с именем *filename*. Возвращает словарь, отображающий расширения имен файлов в строки с названиями типов MIME. Если указанный файл не существует или не может быть прочитан, возвращает None.

`add_type(`*`type`*`, `*`ext`* `[, ` *`strict`*`])`

Добавляет в отображение новый тип MIME. В аргументе *type* передается тип MIME, такой как 'text/plain', в аргументе *ext* – расширение имени файла, такое как '.txt', и в аргументе *strict* – логический флаг, который указывает, является ли данный тип MIME официально зарегистрированным. По умолчанию принимает значение True.

Модуль quopri

Модуль quopri обеспечивает преобразование строк байтов в формат 'quoted-printable' и обратно. Этот формат в основном используется для кодирования текста с 8-битовыми символами, большая часть которых читается при использовании кодировки ASCII, но среди которых может содержаться небольшое количество непечатаемых или специальных символов (например, символы в диапазоне 128-255). Ниже перечислены правила, которые применяются при преобразовании символов в формат 'quoted-printable':

- Любой печатаемый, не пробельный символ ASCII, за исключением '=', представляет сам себя.
- Символ '=' используется как экранирующий символ. Когда за ним следуют две шестнадцатеричные цифры, они представляют символ с указанным значением (например, '=0C'). Сам знак равенства представляется, как '=3D'. Если символ '=' появляется в конце строки, он обозначает мягкий перенос строки. Такой перенос используется, только если длинную входную строку следует разбить на несколько выходных строк.
- Пробелы и табуляции остаются без изменений, но в конце строки они могут удаляться.

Этот формат часто используется для оформления документов, содержащих специальные символы из расширенного набора символов ASCII. Например, если документ содержит текст «Copyright © 2009», в языке Python его можно было бы представить в виде строки байтов `b'Copyright \xa9 2009'`. Та же строка в формате 'quoted-printable' выглядит как `b'Copyright =A9 2009'`, где специальный символ '\xa9' был замещен экранированной последовательностью '=A9'.

`decode(`*`input`*`, `*`output`*` [, `*`header`*`])`

Декодирует данные в формате 'quoted-printable'. В аргументах *input* и *output* передаются объекты, похожие на файлы, открытые в двоичном режиме. Если в аргументе *header* передается значение `True`, то символ подчеркивания (_) будет интерпретироваться как пробел. В противном случае он будет оставаться без изменений. Эту особенность можно использовать при преобразовании заголовков MIME. По умолчанию принимает значение `False`.

`decodestring(`*`s`*` [, `*`header`*`])`

Декодирует строку *s* с данными в формате 'quoted-printable'. В аргументе *s* можно передать строку Юникода или строку байтов, но результат всегда будет строкой байтов. Аргумент *header* имеет тот же смысл, что и в функции `decode()`.

`encode(`*`input`*`, `*`output`*`, `*`quotetabs`*` [, `*`header`*`])`

Преобразует данные в формат 'quoted-printable'. В аргументах *input* и *output* передаются объекты, похожие на файлы, открытые в двоичном режиме. Если в аргументе *quotetabs* передается значение `True`, в дополнение к обычным правилам символы табуляции также будут замещаться экранированными последовательностями. В противном случае они останутся в неизменном виде. По умолчанию аргумент *quotetabs* принимает значение `False`. Аргумент *header* имеет тот же смысл, что и в функции `decode()`.

`encodestring(`*`s`*` [, `*`quotetabs`*` [, `*`header`*`]])`

Преобразует строку байтов *s* в формат 'quoted-printable'. Результат всегда является строкой байтов. Аргументы *quotetabs* и *header* имеют тот же смысл, что и в функции `encode()`.

Примечания

Формат `quoted-printable` появился раньше Юникода и может применяться только для представления 8-битовых данных. Хотя обычно этот формат используется

для преобразования текста, в действительности он применяется только к символам ASCII и к символам из расширенного набора ASCII, которые могут быть представлены одиночными байтами. При использовании этого модуля убедитесь, что файлы открыты в двоичном режиме, а строковые значения передаются в виде строк байтов.

Пакет xml

В состав стандартной библиотеки языка Python входят несколько модулей, предназначенных для обработки данных в формате XML. Тема обработки XML-документов слишком большая, и ее обсуждение выходит далеко за рамки этой книги. В этом разделе предполагается, что читатель уже знаком с основными понятиями, касающимися XML. Для изучения основ XML можно порекомендовать книги, такие как «Inside XML» Стива Холзнера (Steve Holzner) (New Riders) или «XML in a Nutshell» Эллиотта Гароль-да (Elliotte Harold) и У. Скотта Менса (W. Scott Means) (O'Reilly and Associates). Существует несколько книг, посвященных обработке XML на языке Python, включая «Python & XML» Кристофера Джонса (Christopher Jones) (O'Reilly and Associates) и «XML Processing with Python» Сина Макграта (Sean McGrath) (Prentice Hall).

Язык Python обеспечивает два вида поддержки формата XML. Первый тип – базовая поддержка двух основных стандартов парсинга XML, SAX и DOM. Стандарт SAX (Simple API for XML – простой прикладной интерфейс для работы с форматом XML) основан на обработке событий, когда выполняется чтение XML-документа, и по мере того как встречаются элементы разметки XML, вызываются функции-обработчики, выполняющие обработку этих элементов. Стандарт DOM (Document Object Model – объектная модель документов) основывается на представлении XML-документа в виде древовидной структуры. После построения дерева документа предоставляется возможность с помощью интерфейса DOM выполнять обход элементов дерева и извлекать данные. Ни SAX, ни DOM не являются интерфейсами, свойственными исключительно языку Python. В языке Python просто реализован стандартный программный интерфейс, разработанный ранее для языков Java и JavaScript.

Вы можете реализовать обработку XML-документов в своих программах, используя интерфейсы SAX и DOM, однако наиболее удобным программным интерфейсом в стандартной библиотеке является интерфейс ElementTree. Этот интерфейс реализует характерный для Python подход к обработке XML-документов, использующий все преимущества языка Python, и который большинство пользователей считают более простым и более быстрым по сравнению с SAX или DOM. В оставшейся части раздела будут рассматриваться все три подхода к обработке XML-документов, но интерфейс ElementTree будет рассматриваться особенно подробно.

Основное внимание в этом разделе будет уделяться только синтаксическому анализу данных в формате XML. Однако в состав стандартной библио-

теки языка Python входят модули, позволяющие реализовать новые типы парсеров, создавать новые документы XML с самого начала и так далее. Кроме того, существует множество сторонних расширений, добавляющих дополнительные возможности, такие как поддержка языков XSLT и XPATH. Ссылки на страницы с дополнительной информацией можно найти по адресу *http://wiki.python.org/moin/PythonXml*.

Пример документа XML

Ниже приводится пример типичного документа XML, в данном случае – описание рецепта.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<recipe>
    <title>
    Famous Guacamole
    </title>
    <description>
    A southwest favorite!
    </description>
    <ingredients>
        <item num="4"> Large avocados, chopped </item>
        <item num="1"> Tomato, chopped </item>
        <item num="1/2" units="C"> White onion, chopped </item>
        <item num="2" units="tbl"> Fresh squeezed lemon juice </item>
        <item num="1"> Jalapeno pepper, diced </item>
        <item num="1" units="tbl"> Fresh cilantro, minced </item>
        <item num="1" units="tbl"> Garlic, minced </item>
        <item num="3" units="tsp"> Salt </item>
        <item num="12" units="bottles"> Ice-cold beer </item>
    </ingredients>
    <directions>
    Combine all ingredients and hand whisk to desired consistency.
    Serve and enjoy with ice-cold beers.
    </directions>
</recipe>
```

Документ состоит из *элементов*, которые начинаются и оканчиваются тегами, например: `<title>...</title>`. Обычно элементы могут вкладываться друг в друга и образовывать иерархии, например элементы `<item>` вкладываются в элемент `<ingredients>`. В каждом XML-документе существует единственный элемент, который называется корневым элементом документа. В данном примере это элемент `<recipe>`. Элементы могут иметь дополнительные атрибуты, как, например, элементы `item` в примере: `<item num="4">Large avocados, chopped</item>`.

При работе с XML-документами обычно приходится иметь дело со всеми этими основными особенностями. Например, может потребоваться извлекать текст и атрибуты из элементов определенных типов. Чтобы отыскать требуемые элементы, необходимо выполнить обход дерева документа, начиная с корневого элемента.

Модуль xml.dom.minidom

Модуль `xml.dom.minidom` обеспечивает поддержку парсинга XML-документов и сохранение результатов в памяти, в виде древовидной структуры, соответствующей стандарту DOM. **В модуле имеется две функции, выполняющие парсинг:**

`parse(`*`file`* `[,` *`parser`*`])`

Анализирует содержимое файла *`file`* и возвращает узел, представляющий вершину дерева документа. В аргументе *`file`* передается имя файла или объект файла, открытый для чтения. В необязательном аргументе *`parser`* можно передать **SAX2-совместимый объект парсера, который будет использоваться** для конструирования дерева. Если этот аргумент опущен, будет использоваться парсер по умолчанию.

`parseString(`*`string`* `[,` *`parser`*`])`

То же, что и `parse()`, за исключением того, что входные данные передаются в виде строки, а не в виде файла.

Узлы

Дерево документа, возвращаемое функциями парсинга, представляет собой коллекцию узлов, связанных между собой. Каждый объект *`n`*, представляющий узел, обладает следующими атрибутами, которые могут использоваться для извлечения информации и при обходе дерева:

Атрибут узла	Описание
`n``.attributes`	Объект отображения, хранящий значения атрибутов (если имеются).
`n``.childNodes`	Список всех дочерних узлов для узла *`n`*.
`n``.firstChild`	Первый дочерний узел для узла *`n`*.
`n``.lastChild`	Последний дочерний узел для узла *`n`*.
`n``.localName`	Локальное имя элемента. Если в имени тега присутствует двоеточие (например, '`<foo:bar ...>`'), в этом атрибуте будет находиться часть имени, следующая за двоеточием.
`n``.namespaceURI`	Пространство имен, ассоциированное с узлом *`n`* (если имеется).
`n``.nextSibling`	Узел, следующий за данным узлом *`n`* в дереве и относящийся к тому же родительскому узлу. Если узел является последним среди узлов того же уровня, этот атрибут получает значение `None`.
`n``.nodeName`	Имя узла. Смысл зависит от типа узла.
`n``.nodeType`	Целое число, описывающее тип узла. Принимает одно из следующих значений, которые являются атрибутами класса `Node`: `ATTRIBUTE_NODE`, `CDATA_SECTION_NODE`, `COMMENT_NODE`, `DOCUMENT_FRAGMENT_NODE`, `DOCUMENT_NODE`, `DOCUMENT_TYPE_NODE`, `ELEMENT_NODE`, `ENTITY_NODE`, `ENTITY_REFERENCE_NODE`, `NOTATION_NODE`, `PROCESSING_INSTRUCTION_NODE` или `TEXT_NODE`.

Атрибут узла	Описание
n.nodeValue	Значение узла. Смысл зависит от типа узла.
n.parentNode	Ссылка на родительский узел.
n.prefix	Часть имени тега, предшествующая двоеточию. Например, для элемента '<foo:bar ...>' этот атрибут будет содержать значение 'foo'.
n.previousSibling	Узел, предшествующий данному узлу *n* в дереве и относящийся к тому же родительскому узлу.

Кроме этих атрибутов, все узлы обладают следующими методами. Обычно они используются для манипулирования структурой дерева.

n.appendChild(*child*)

Добавляет новый дочерний узел *child* в узел *n*. Новый узел добавляется после последнего дочернего узла.

n.cloneNode(*deep*)

Создает копию узла *n*. Если в аргументе *deep* передается значение True, все дочерние узлы также будут скопированы.

n.hasAttributes()

Возвращает True, если узел имеет какие-либо атрибуты.

n.hasChildNodes()

Возвращает True, если узел имеет какие-либо дочерние узлы.

n.insertBefore(*newchild*, *ichild*)

Вставляет новый дочерний узел *newchild* перед другим дочерним узлом *ichild*. Узел *ichild* уже должен быть дочерним узлом для *n*.

n.isSameNode(*other*)

Возвращает True, если узел *other* ссылается на тот же узел DOM, что и *n*.

n.normalize()

Объединяет смежные текстовые узлы в один текстовый узел.

n.removeChild(*child*)

Удаляет дочерний узел *child* из узла *n*.

n.replaceChild(*newchild*,*oldchild*)

Замещает дочерний узел *oldchild* узлом *newchild*. Узел *oldchild* должен быть дочерним узлом для узла *n*.

Несмотря на многообразие типов узлов, которые могут присутствовать в дереве документа, все-таки наиболее часто приходится работать с узлами типа Document, Element и Text. Далее коротко описывается каждый из этих типов.

Узлы типа Document

Узел *d* типа `Document` располагается на вершине дерева документа и представляет документ целиком. Он обладает следующими атрибутами и методами:

d.`documentElement`

Содержит корневой элемент всего документа.

d.`getElementsByTagName(`*tagname*`)`

Отыскивает все дочерние узлы и возвращает список элементов с именами тегов *tagname*.

d.`getElementsByTagNameNS(`*namespaceuri*, *localname*`)`

Отыскивает все дочерние узлы и возвращает список элементов, соответствующих указанному пространству имен *namespaceuri* и локальному имени *localname*. Возвращаемый список является объектом класса `NodeList`.

Узлы типа Element

Узел *e* типа `Element` представляет одиночный элемент XML, такой как '`<foo>...</foo>`'. Чтобы извлечь текст из элемента, необходимо отыскать в нем дочерние узлы типа `Text`. Для извлечения другой информации могут использоваться следующие атрибуты и методы:

e.`tagName`

Имя тега элемента. Например, если элемент определен как '`<foo ...>`', именем тега будет строка '`foo`'.

e.`getElementsByTagName(`*tagname*`)`

Возвращает список всех дочерних узлов с указанным именем тега.

e.`getElementsByTagNameNS(`*namespaceuri*, *localname*`)`

Возвращает список всех дочерних узлов с указанным именем тега в указанном пространстве имен. В аргументах *namespaceuri* и *localname* передаются строки, определяющие пространство имен и локальное имя тега соответственно. Если пространство имен объявлено, как '`<foo xmlns:foo="http://www.spam.com/foo">`', в аргументе *namespaceuri* должна передаваться строка '`http://www.spam.com/foo`'. При поиске последующих элементов '`<foo:bar>`' в аргументе *localname* должна передаваться строка '`bar`'. Возвращаемый список является объектом класса `NodeList`.

e.`hasAttribute(`*name*`)`

Возвращает `True`, если элемент имеет атрибут с именем *name*.

e.`hasAttributeNS(`*namespaceuri*, *localname*`)`

Возвращает `True`, если элемент имеет атрибут с именем, включающим пространство имен *namespaceuri* и локальное имя *localname*. Аргументы имеют тот же смысл, что и в методе `getElementsByTagNameNS()`.

e.`getAttribute(`*name*`)`

Возвращает значение атрибута *name*. Возвращаемое значение является строкой. Если искомый атрибут не существует, возвращается пустая строка.

e.getAttributeNS(*namespaceuri*, *localname*)

Возвращает значение атрибута с именем, включающим пространство имен *namespaceuri* и локальное имя *localname*. Возвращаемое значение является строкой. **Если искомый атрибут не существует, возвращается пустая строка.** Аргументы имеют тот же смысл, что и в методе getElementsByTagNameNS().

Узлы типа Text

Узлы типа Text используются для представления текстовых данных. Текстовые данные сохраняются в атрибуте *t*.data объекта *t* типа Text. Текст, ассоциированный с тем или иным элементом документа, всегда хранится в дочерних узлах типа Text.

Вспомогательные методы

Ниже перечислены вспомогательные методы узлов. Они не являются частью стандартного интерфейса DOM и предоставляются языком Python для удобства и для использования в отладочных целях.

n.toprettyxml([*indent* [, *newl*]])

Создает отформатированную разметку XML, представляющую узел *n* и его дочерние узлы. Аргумент *indent* определяет строку, которая будет использоваться при оформлении отступов. По умолчанию принимает значение '\t'. Аргумент *newl* определяет символ перевода строки и по умолчанию имеет значение '\n'.

n.toxml([*encoding*])

Создает разметку XML, представляющую узел *n* и его дочерние узлы. Аргумент *encoding* определяет название кодировки (например, 'utf-8'). Если кодировка не указана, в выходном тексте она не указывается.

n.writexml(*writer* [, *indent* [, *addindent* [, *newl*]]])

Записывает разметку XML в объект *writer*. В аргументе *writer* можно передать любой объект, реализующий метод write(), совместимый с интерфейсом файлов. Аргумент *indent* определяет строку, которая будет использоваться при оформлении отступов. Эта строка будет добавляться перед началом вывода узла *n*. Аргумент *addindent* определяет строку, которая будет использоваться при оформлении отступов во время вывода дочерних узлов узла *n*. Аргумент *newl* определяет символ перевода строки.

Пример использования интерфейса DOM

Следующий пример демонстрирует использование модуля xml.dom.minidom для анализа и получения информации из XML-файла:

```
from xml.dom import minidom
doc = minidom.parse("recipe.xml")

ingredients = doc.getElementsByTagName("ingredients")[0]
items       = ingredients.getElementsByTagName("item")

for item in items:
    num      = item.getAttribute("num")
```

```
units    = item.getAttribute("units")
text     = item.firstChild.data.strip()
quantity = "%s %s" % (num,units)
print("%-10s %s" % (quantity,text))
```

Примечание

Модуль `xml.dom.minidom` имеет множество других возможностей, позволяющих влиять на процесс парсинга дерева и работать с другими типами узлов XML. Дополнительную информацию по этой теме можно найти в электронной документации.

Модуль xml.etree.ElementTree

Модуль `xml.etree.ElementTree` содержит определение гибкого контейнерного класса `ElementTree`, предназначенного для хранения иерархических данных и манипулирования ими. Несмотря на то что объекты этого класса часто используются для работы с документами XML, тем не менее область применения этих объектов значительно шире – фактически они совмещают в себе возможности словаря и списка.

Объекты класса ElementTree

Класс `ElementTree` используется для создания нового объекта, представляющего верхний уровень иерархии.

`ElementTree([`*`element`* `[,` *`file`*`]])`

Создает новый объект класса `ElementTree`. В аргументе *`element`* передается экземпляр, представляющий корневой узел дерева. Этот экземпляр поддерживает интерфейс элементов, описанный ниже. В аргументе *`file`* передается имя файла или объект файла, откуда будут читаться данные в формате XML для заполнения дерева.

Экземпляр *`tree`* класса `ElementTree` обладает следующими методами:

`tree``._setroot(`*`element`*`)`

Назначает *`element`* корневым элементом.

`tree``.find(`*`path`*`)`

Отыскивает и возвращает первый элемент верхнего уровня в дереве, соответствующий критерию в аргументе *`path`*. В аргументе *`path`* передается строка, описывающая тип элемента и его положение относительно других элементов. Синтаксис аргумента *`path`* описывается в следующем списке:

Значение path	Описание
'*tag*'	Соответствует только элементам верхнего уровня с именем тега *tag*, например <*tag*>...</*tag*>. Не совпадает с элементами на более низких уровнях. Элемент типа *tag*, находящийся внутри другого элемента, например <foo><*tag*>...</*tag*></foo>, не соответствует данному значению аргумента *path*.

Значение path	Описание
'*parent/tag*'	Соответствует элементам с именем тега *tag*, если они являются дочерними элементами для элементов с тегом '*parent*'. Количество компонентов в аргументе *path* не ограничивается.
'*'	Соответствует любым дочерним элементам. Например, значению '*/tag' соответствуют все дочерние элементы с именем тега 'tag'.
'.'	Начинает поиск от текущего узла.
'//'	Соответствует всем вложенным элементам на всех уровнях, ниже уровня указанного элемента. Например, значению './/tag' соответствуют все вложенные элементы с именем 'tag', на всех вложенных уровнях.

При работе с документами XML, в которых используются пространства имен, элемент '*tag*' в аргументе *path* должен иметь вид '{*uri*}*tag*', где *uri* – это строка, такая как 'http://www.w3.org/TR/html4/'.

tree.findall(*path*)

Отыскивает все элементы верхнего уровня в дереве, соответствующие критерию в аргументе *path*, и возвращает их в том порядке, в каком они следуют в документе, в виде списка или итератора.

tree.findtext(*path* [, *default*])

Возвращает текстовый элемент для первого элемента верхнего уровня в дереве, соответствующего критерию в аргументе *path*. В аргументе *default* передается строка, которая будет возвращена, если в дереве документа не найдется элемента, соответствующего условию.

tree.getiterator([*tag*])

Создает итератор, который воспроизводит все элементы в дереве, в порядке их следования, имена тегов которых совпадают со значением аргумента *tag*. Если аргумент *tag* опущен, будут возвращены все элементы, присутствующие в дереве.

tree.getroot()

Возвращает корневой элемент дерева.

tree.parse(*source* [, *parser*])

Выполняет парсинг внешнего XML-документа и замещает корневой элемент полученным результатом. В аргументе *source* передается имя файла или объект, похожий на файл, представляющий XML-документ. В необязательном аргументе *parser* можно передать экземпляр класса TreeBuilder, который описывается ниже.

tree.write(*file* [, *encoding*])

Записывает все содержимое дерева в файл. В аргументе *file* передается имя файла или объект, похожий на файл, открытый для записи. В аргументе *encoding* передается название кодировки, которая будет использоваться

при выводе данных. Если этот аргумент опущен, используется кодировка, применяемая интерпретатором по умолчанию (в большинстве случаев 'utf-8' или 'ascii').

Создание элементов

Элементы, которые могут храниться в контейнере класса ElementTree, являются экземплярами различных классов, которые создаются либо в процессе парсинга содержимого файла, либо с помощью следующих функций конструкторов:

Comment([*text*])

Создает новый элемент комментария. В аргументе *text* передается строка или строка байтов с текстом. При выводе этот элемент отображается в комментарий XML.

Element(*tag* [, *attrib* [, ***extra*]])

Создает новый элемент. В аргументе *tag* передается имя тега элемента. Например, чтобы создать элемент '<foo>...</foo>', в аргументе *tag* следует передать строку 'foo'. В аргументе *attrib* передается словарь атрибутов элемента, определяемых как строки или строки байтов. Любые дополнительные именованные аргументы в *extra* также используются для установки значений атрибутов элемента.

fromstring(*text*)

Создает элемент из фрагмента разметки XML в аргументе *text*; так же, как и функция XML(), которая описывается ниже.

ProcessingInstruction(*target* [, *text*])

Создает новый элемент, соответствующий инструкции обработки. В аргументах *target* и *text* передаются строки или строки байтов. При отображении в разметку XML этот элемент соответствует тексту '<?*target text*?>'.

SubElement(*parent*, *tag* [, *attrib* [, ***extra*]])

То же, что и Element(), но автоматически добавляет новый элемент, как дочерний, в элемент *parent*.

XML(*text*)

Создает элемент из фрагмента разметки XML в аргументе *text*. Например, если в аргументе *text* передать строку '<foo>...</foo>', эта функция создаст стандартный элемент с тегом 'foo'.

XMLID(*text*)

То же, что и XML(*text*), за исключением того, что собирает все атрибуты 'id' и создает словарь, отображающий значения атрибутов 'id' в элементы. Возвращает кортеж (*elem*, *idmap*), где в поле *elem* содержится новый элемент, а в поле *idmap* – словарь, отображающий значения атрибутов ID. Например, вызов XMLID('<foo id="123"><bar id="456">Hello</bar></foo>') вернет (<Element foo>, {'123': <Element foo>, '456': <Element bar>}).

Интерфейс элементов

Несмотря на то, что объекты элементов, хранящиеся в контейнере типа ElementTree, могут быть экземплярами разных классов, тем не менее все они имеют общий интерфейс. Элемент *elem* любого типа поддерживает следующие операции языка Python:

Операция	Описание
elem[*n*]	Возвращает *n*-й дочерний элемент в элементе *elem*.
elem[*n*] = *newelem*	Замещает *n*-й дочерний элемент в элементе *elem* новым элементом *newelem*.
del *elem*[*n*]	Удаляет *n*-й дочерний элемент из элемента *elem*.
len(*elem*)	Возвращает количество дочерних элементов в элементе *elem*.

Все элементы обладают следующими атрибутами:

Атрибут	Описание
elem.tag	Строка, определяющая тип элемента. Например, для элемента <foo>...</foo> атрибут tag будет иметь значение 'foo'.
elem.text	Данные, ассоциированные с элементом. Обычно – строка, содержащая текст, заключенный между открывающим и закрывающим тегами элемента XML.
elem.tail	Дополнительные данные, ассоциированные с элементом. В случае XML в этом атрибуте обычно сохраняется текст, следующий после закрывающего тега элемента, но перед следующим тегом.
elem.attrib	Словарь с атрибутами элемента.

Элементы поддерживают следующие методы, часть которых имитируют методы словарей:

elem.append(*subelement*)

Добавляет элемент *subelement* в список дочерних элементов.

elem.clear()

Стирает всю информацию, имеющуюся в элементе, включая атрибуты, текст и дочерние элементы.

elem.find(*path*)

Отыскивает первый вложенный элемент, соответствующий критерию в аргументе *path*.

elem.findall(*path*)

Отыскивает все вложенные элементы, соответствующие критерию в аргументе *path*. Возвращает список или итерируемый объект с элементами, следующими в том же порядке, в каком они присутствуют в документе.

elem.findtext(*path* [, *default*])

Отыскивает текст для первого элемента, соответствующего критерию в аргументе *path*. В аргументе *default* передается строка, которая возвращается в случае, если метод не найдет совпадений.

elem.get(*key* [, *default*])

Извлекает значение атрибута *key*. В аргументе *default* передается значение, которое возвращается в случае, если искомый атрибут отсутствует. Если в документе используются пространства имен XML, в аргументе *key* должна передаваться строка вида '{*uri*}*key*', где в поле *uri* передается строка, такая как 'http://www.w3.org/TR/html4/'.

elem.getchildren()

Возвращает все вложенные элементы в порядке следования в документе.

elem.getiterator([*tag*])

Возвращает итератор, который воспроизводит все вложенные элементы с тегом *tag*.

elem.insert(*index*, *subelement*)

Вставляет вложенный элемент в список дочерних элементов, в позицию *index*.

elem.items()

Возвращает все атрибуты элемента в виде списка кортежей (*name*, *value*).

elem.keys()

Возвращает список имен всех атрибутов.

elem.remove(*subelement*)

Удаляет элемент *subelement* из списка дочерних элементов.

elem.set(*key*, *value*)

Записывает значение *value* в атрибут *key*.

Построение дерева

Объект класса ElementTree легко создается из других древовидных структур. Для этой цели можно использовать объекты следующего класса.

TreeBuilder([*element_factory*])

Класс, позволяющий создать структуру типа ElementTree **с помощью последовательности вызовов** start(), end() и data() **в процессе анализа содержимого файла или во время обхода другой древовидной структуры. В аргументе** *element_factory* передается функция, которая будет вызываться для создания новых экземпляров элементов.

Экземпляр *t* класса TreeBuilder обладает следующими методами:

t.close()

Закрывает построитель дерева и возвращает созданный объект ElementTree верхнего уровня.

t.data(*data*)

Добавляет текстовые данные в текущий обрабатываемый элемент.

t.end(*tag*)

Закрывает текущий обрабатываемый элемент и возвращает окончательный объект элемента.

t.start(*tag*, *attrs*)

Создает новый элемент. В аргументе *tag* передается имя элемента, а в аргументе *attrs* – словарь со значениями атрибутов.

Вспомогательные функции

Ниже перечислены вспомогательные функции, объявленные в модуле xml.etree.ElementTree:

dump(*elem*)

Выводит структуру элемента *elem* в поток sys.stdout для отладки. Вывод обычно имеет вид разметки XML.

iselement(*elem*)

Проверяет, является ли элемент *elem* допустимым объектом элемента.

iterparse(*source* [, *events*])

Выполняет парсинг очередного фрагмента XML с приращением, извлекая его из *source*. В аргументе *source* передается имя файла или объект, похожий на файл, содержащий данные в формате XML. В аргументе *events* передается список типов возбуждаемых событий. К допустимым относятся типы событий: 'start', 'end', 'start-ns' и 'end-ns'. Если этот аргумент опущен, возбуждаться будет только событие 'end'. Функция возвращает итератор, который воспроизводит кортежи (*event*, *elem*), где поле *event* содержит строку, такую как 'start' или 'end', а поле *elem* – обрабатываемый элемент. В случае события 'start' в кортеже возвращается только что созданный элемент, изначально пустой, за исключением атрибутов. В случае события 'end' возвращается полностью заполненный элемент, включая все вложенные элементы.

parse(*source*)

Полностью преобразует исходную разметку XML в объект ElementTree. В аргументе *source* передается имя файла или объект, похожий на файл, с данными в формате XML.

tostring(*elem*)

Создает строку XML, представляющую элемент *elem* и все вложенные в него элементы.

Примеры обработки XML-документов

Ниже приводится пример использования класса ElementTree для парсинга XML-файла с рецептом и вывода списка ингредиентов. Он напоминает реализацию с использованием интерфейса DOM.

```
from xml.etree.ElementTree import ElementTree

doc = ElementTree(file="recipe.xml")
ingredients = doc.find('ingredients')

for item in ingredients.findall('item'):
    num      = item.get('num')
    units    = item.get('units','')
    text     = item.text.strip()
    quantity = "%s %s" % (num, units)
    print("%-10s %s" % (quantity, text))
```

Синтаксис путей, используемый методами объекта `ElementTree`**, облегчает и упрощает решение некоторых задач. Например, ниже приводится другая версия предыдущего примера, где использование синтаксиса путей позволило упростить извлечение всех элементов** `<item>...</item>`**.**

```
from xml.etree.ElementTree import ElementTree

doc = ElementTree(file="recipe.xml")
for item in doc.findall(".//item"):
    num      = item.get('num')
    units    = item.get('units','')
    text     = item.text.strip()
    quantity = "%s %s" % (num, units)
    print("%-10s %s" % (quantity, text))
```

Взгляните на следующий пример XML-файла `'recipens.xml'`**, в котором используются пространства имен:**

```
<?xml version="1.0" encoding="iso-8859-1"?>
<recipe xmlns:r="http://www.dabeaz.com/namespaces/recipe">
    <r:title>
    Famous Guacamole
    </r:title>
    <r:description>
    A southwest favorite!
    </r:description>
    <r:ingredients>
        <r:item num="4"> Large avocados, chopped </r:item>
        ...
    </r:ingredients>
    <r:directions>
    Combine all ingredients and hand whisk to desired consistency.
    Serve and enjoy with ice-cold beers.
    </r:directions>
</recipe>
```

При работе с пространствами имен проще всего использовать словарь, который отображает префиксы пространств имен в соответствующие им идентификаторы URI. Благодаря этому можно использовать операторы форматирования строк для подстановки URI, как показано ниже:

```
from xml.etree.ElementTree import ElementTree
doc = ElementTree(file="recipens.xml")
ns = {
```

```
    'r' : 'http://www.dabeaz.com/namespaces/recipe'
}
ingredients = doc.find('{%(r)s}ingredients' % ns)
for item in ingredients.findall('{%(r)s}item' % ns):
    num      = item.get('num')
    units    = item.get('units','')
    text     = item.text.strip()
    quantity = "%s %s" % (num, units)
    print("%-10s %s" % (quantity, text))
```

Модуль ElementTree отлично подходит для работы с небольшими файлами XML, позволяя быстро загружать их в память и работать с ними. Однако допустим, что необходимо обработать огромный файл XML со следующей структурой:

```
<?xml version="1.0" encoding="utf-8"?>
<music>
    <album>
        <title>A Texas Funeral</title>
        <artist>Jon Wayne</artist>
        ...
    </album>
    <album>
        <title>Metaphysical Graffiti</title>
        <artist>The Dead Milkmen</artist>
        ...
    </album>
    ... далее следуют описания еще 100000 альбомов ...
</music>
```

Чтобы прочитать такой огромный файл целиком, потребуется значительный объем памяти. Например, файл XML объемом 10 Мбайт может занять более 100 Мбайт памяти. Если необходимо извлечь информацию из такого огромного файла, проще всего будет использовать функцию ElementTree.iterparse(). Ниже приводится пример реализации итеративной обработки узлов <album> из предыдущего файла:

```
from xml.etree.ElementTree import iterparse

iparse = iterparse("music.xml", ['start','end'])
# Отыскать элемент верхнего уровня music
for event, elem in iparse:
    if event == 'start' and elem.tag == 'music':
        musicNode = elem
        break

# Получить все альбомы
albums = (elem for event, elem in iparse
              if event == 'end' and elem.tag == 'album')

for album in albums:
    # Обработать очередной альбом
    ...
    musicNode.remove(album) # По завершении удалить альбом из памяти
```

Ключом к эффективному использованию функции `iterparse()` является удаление данных, надобность в которых отпала. Последняя инструкция `musicNode.remove(album)` в примере удаляет очередной элемент `<album>` после его обработки (удаляя его из родительского элемента). Если проверить, какой объем памяти потребляет предыдущая программа в процессе выполнения, можно обнаружить, что он совсем невелик, хотя она обрабатывает значительный объем данных.

Примечания

- Модуль `ElementTree` обеспечивает чрезвычайно простой и гибкий способ обработки простых XML-документов на языке Python. Однако он не настолько богат возможностями, как хотелось бы. Например, он не предоставляет средств проверки корректности разметки XML и не поддерживает возможность обработки некоторых сложных аспектов XML-документов, таких как объявления DTD. Для этого необходимо устанавливать и использовать сторонние пакеты. Один из таких пакетов, `lxml.etree` (*http://codespeak.net/lxml/*), реализует программный интерфейс `ElementTree` к популярным библиотекам `libxml2` и `libxslt` и обеспечивает полную поддержку языков XPATH, XSLT и других возможностей.
- Модуль `ElementTree` сам по себе является сторонним пакетом, поддержкой которого занимается Фредерик Лунд (Fredrik Lundh) (домашняя страница проекта: *http://effbot.org/zone/element-index.htm*). На этом сайте можно найти более современные версии по сравнению с той, что включена в состав стандартной библиотеки, и предлагающие дополнительные возможности.

Модуль xml.sax

Модуль `xml.sax` обеспечивает возможность парсинга XML-документов, предоставляя интерфейс SAX2.

`parse(`*`file`*`, `*`handler`*` [, `*`error_handler`*`])`

Выполняет парсинг XML-документа *`file`*. В аргументе *`file`* передается имя файла или открытый объект файла. В аргументе *`handler`* передается объект, выполняющий обработку содержимого документа. В необязательном аргументе *`error_handler`* передается объект, реализующий обработку ошибок, описание которого можно найти в электронной документации.

`parseString(`*`string`*`, `*`handler`*` [, `*`error_handler`*`])`

То же, что и `parse()`, но выполняет парсинг XML-данных в строке *`string`*.

Объекты-обработчики

Чтобы выполнить какую-либо обработку XML-документа, функции `parse()` или `parseString()` требуется передать объект-обработчик. Чтобы объявить обработчик, необходимо определить класс, производный от класса `ContentHandler`. Экземпляр *`c`* класса `ContentHandler` обладает следующими методами, каждый из которых можно переопределить в своем классе обработчика:

c.characters(*content*)

Вызывается парсером, который передает исходные текстовые данные. В аргументе *content* передается строка символов.

c.endDocument()

Вызывается парсером по достижении конца документа.

c.endElement(*name*)

Вызывается парсером по достижении конца элемента с именем *name*. Например, по достижении закрывающего тега '</foo>' парсер вызывает этот метод и передает ему строку 'foo' в аргументе *name*.

c.endElementNS(*name*, *qname*)

Вызывается по достижении конца элемента, в теге которого используется пространство имен XML. В аргументе *name* передается кортеж строк (*uri*, *localname*), а в аргументе *qname* – полностью квалифицированное имя тега. Обычно в аргументе *qname* передается значение None, если поддержка префиксов имен в SAX не была включена. Например, если элемент определен как '<foo:bar xmlns:foo="http://spam.com">', в аргументе *name* будет передан кортеж (u'http://spam.com', u'bar').

c.endPrefixMapping(*prefix*)

Вызывается по достижении пространства имен XML. В аргументе *prefix* передается имя пространства имен.

c.ignorableWhitespace(*whitespace*)

Вызывается при встрече в документе пустого пространства, которое должно игнорироваться парсером. В аргументе *whitespace* передается строка, содержащая пробельные символы.

c.processingInstruction(*target*, *data*)

Вызывается, когда парсер встречает инструкцию обработки XML, заключенную в конструкцию <? ... ?>. В аргументе *target* передается тип инструкции, а в аргументе *data* – данные инструкции. Например, для инструкции '<?xml-stylesheet href="mystyle.css" type="text/css"?>' в аргументе *target* будет передана строка 'xml-stylesheet', а в аргументе *data* – остаток текста инструкции 'href="mystyle.css" type="text/css"'.

c.setDocumentLocator(*locator*)

Вызывается парсером, чтобы передать объект с информацией о местоположении, который может использоваться для отслеживания номеров строк, символов в строке и получения другой информации. Основное назначение этого метода состоит в том, чтобы просто сохранить где-нибудь объект с информацией о местоположении и обеспечить тем самым возможность воспользоваться им позднее, например когда потребуется вывести сообщение об ошибке. Объект с информацией о местоположении, который передается в аргументе *locator*, предоставляет четыре метода: getColumnNumber(), getLineNumber(), getPublicId() и getSystemId(), которые могут использоваться для получения необходимой информации.

c.skippedEntity(*name*)

Вызывается, когда парсер встречает мнемонику. В аргументе *name* передается имя мнемоники.

c.startDocument()

Вызывается в начале документа.

c.startElement(*name*, *attrs*)

Вызывается, когда парсер встречает новый элемент XML. В аргументе *name* передается имя элемента, а в аргументе *attrs* – объект с информацией об атрибутах. Например, для XML-элемента '<foo bar="whatever" spam="yes">', в аргументе *name* метод получит строку 'foo', а в аргументе *attrs* – объект с информацией об атрибутах bar и spam. Объект *attrs* обладает следующими методами, которые могут использоваться для получения информации об атрибутах:

Метод	Описание
attrs.getLength()	Возвращает количество атрибутов
attrs.getNames()	Возвращает список имен атрибутов
attrs.getType(name)	Возвращает тип атрибута *name*
attrs.getValue(name)	Возвращает значение атрибута *name*

c.startElementNS(*name*, *qname*, *attrs*)

Вызывается, когда парсер встречает новый элемент XML и в документе используются пространства имен. В аргументе *name* передается кортеж строк (*uri*, *localname*), а в аргументе *qname* – полностью квалифицированное имя тега (обычно в аргументе *qname* передается значение None, если поддержка префиксов имен в SAX2 не была включена). В аргументе *attrs* передается объект с информацией об атрибутах. Например, для элемента '<foo:bar xmlns:foo="http://spam.com" blah="whatever">' в аргументе *name* метод получит кортеж (u'http://spam.com', u'bar'), в аргументе *qname* – значение None и в аргументе *attrs* – объект с информацией об атрибуте blah. Объект *attrs* обладает теми же методами, что и объект, который передается методу startElement(), описанному выше. Кроме того, этот объект обладает следующими дополнительными методами, предназначенными для работы с пространствами имен:

Метод	Описание
attrs.getValueByQName(*qname*)	Возвращает значение атрибута с полностью квалифицированным именем *qname*.
attrs.getNameByQName(*qname*)	Возвращает кортеж (*namespace*, *localname*) для атрибута с полностью квалифицированным именем *qname*
attrs.getQNameByName(*name*)	Возвращает полностью квалифицированное имя для атрибута с локальным именем *name*
attrs.getQNames()	Возвращает полностью квалифицированные имена для всех атрибутов.

c.startPrefixMapping(*prefix*, *uri*)

Вызывается в начале объявления пространства имен XML. Например, если элемент определен как '<foo:bar xmlns:foo="http://spam.com">', в аргументе *prefix* будет передана строка 'foo', а в аргументе *uri* — строка 'http://spam.com'.

Пример

Следующий пример иллюстрирует реализацию парсера на основе интерфейса SAX, который выводит список ингредиентов из файла рецепта, представленного ранее. Сравните его с примером из раздела с описанием модуля xml.dom.minidom.

```
from xml.sax import ContentHandler, parse

class RecipeHandler(ContentHandler):
    def startDocument(self):
        self.initem = False
    def startElement(self,name,attrs):
        if name == 'item':
            self.num    = attrs.get('num','1')
            self.units  = attrs.get('units','none')
            self.text   = []
            self.initem = True
    def endElement(self,name):
        if name == 'item':
            text = "".join(self.text)
            if self.units == 'none': self.units = ""
            unitstr = "%s %s" % (self.num, self.units)
            print("%-10s %s" % (unitstr,text.strip()))
            self.initem = False
    def characters(self,data):
        if self.initem:
            self.text.append(data)

parse("recipe.xml",RecipeHandler())
```

Примечания

Модуль xml.sax имеет массу дополнительных возможностей, которые могут использоваться для обработки различных типов XML-документов и создания собственных парсеров. Например, он позволяет определять объекты-обработчики для анализа объявлений DTD и других частей документа. Дополнительную информацию можно найти в электронной документации.

Модуль xml.sax.saxutils

Модуль xml.sax.saxutils содержит несколько вспомогательных функций и объектов, которые часто используются при реализации SAX-парсеров, но также нередко бывают полезны при решении других задач.

escape(*data* [, *entities*])

Замещает специальные символы в строке *data* их экранированными аналогами. Например, символ '<' замещается строкой '<'. В аргументе *entities*

можно передать дополнительный словарь, отображающий символы в экранированные последовательности. Например, если в аргументе *entities* передать словарь { u'\xf1' : 'ñ' }, функция заменит все вхождения символа с строкой 'ñ'.

`unescape(`*data* `[, ` *entities*`])`

Выполняет замещение экранированных последовательностей в строке *data* фактическими символами. Например, последовательность '<' будет замещена символом '<'. В аргументе *entities* можно передать дополнительный словарь, отображающий мнемоники в неэкранированные символы. Словарь *entities* в этой функции является зеркальным отражением словаря, который передается функции `escape()`, например { 'ñ' : u'\xf1' }.

`quoteattr(`*data* `[, ` *entities*`])`

Экранирует специальные символы в строке *data* и при этом выполняет дополнительную обработку, которая позволяет использовать результат в качестве значения атрибута XML. Возвращаемое значение можно выводить непосредственно, как значение атрибута, например `print "<element attr=%s>" % quoteattr(`*somevalue*`)`. В аргументе *entities* передается словарь, пригодный для использования в вызове функции `escape()`.

`XMLGenerator([`*out* `[, ` *encoding*`]])`

Возвращает объект типа `ContentHandler`, который просто выводит данные в формате XML обратно в поток вывода в виде XML-документа. Создает копию оригинального XML-документа. Аргумент *out* определяет поток вывода, по умолчанию вывод выполняется в поток `sys.stdout`. Аргумент *encoding* определяет кодировку символов и по умолчанию принимает значение 'iso-8859-1'. Этот объект удобно использовать при отладке программного кода, выполняющего парсинг и использующего заведомо корректный обработчик.

25

Различные библиотечные модули

Модули, перечисленные в этом разделе, не рассматривались подробно в этой книге, однако они являются частью стандартной библиотеки Python. Эти модули не были описаны главным образом потому, что они либо являются слишком низкоуровневыми, либо имеют ограниченную область применения, либо их применение ограничено определенными платформами, либо устарели, либо они настолько сложные, что для их подробного исследования потребовалось бы написать отдельную книгу. Несмотря на то что описание этих модулей не вошло в книгу, тем не менее описание каждого модуля можно найти в электронной документации, по адресу *http://docs.python.org/library/modname*. Предметный указатель всех модулей можно также найти на странице *http://docs.python.org/library/modindex.html*.

Модули, перечисленные в этой главе, составляют подмножество модулей, общих для Python 2 и Python 3. Если в программе используется какой-то модуль, не перечисленный здесь, есть вероятность, что он официально был признан как нерекомендуемый к использованию. В версии Python 3 имена некоторых модулей были изменены. Там, где это необходимо, новые имена указаны в круглых скобках.

Службы интерпретатора Python

Ниже перечислены модули, обеспечивающие дополнительные возможности, имеющие отношение к языку программирования Python и к особенностям выполнения интерпретатора Python. Многие эти модули имеют отношение к синтаксическому анализу и компиляции исходных текстов программ на языке Python.

Модуль	Описание
`bdb`	Доступ к отладчику
`code`	Базовые классы интерпретатора
`codeop`	Компиляция программного кода на языке Python

(продолжение)

Модуль	Описание
compileall	Компиляция всех файлов с программным кодом на языке Python, присутствующих в каталоге
copy_reg (copyreg)	Регистрирует встроенные типы для использования в модуле pickle
dis	Дизассемблер
distutils	Функции для создания дистрибутивов модулей Python
fpectl	Управление исключениями, возникающими в операциях с плавающей точкой
imp	Доступ к реализации инструкции import
keyword	Проверяет, является ли строка ключевым словом языка Python
linecache	Извлекает строки из исходных файлов
modulefinder	Поиск модулей, используемых сценарием
parser	Доступ к деревьям парсинга исходного программного кода на языке Python
pickletools	Инструменты для разработчиков средств сериализации объектов
pkgutil	Вспомогательные функции, используемые для упаковывания программ
pprint	Форматированный вывод объектов
pyclbr	Извлекает информацию для классов броузеров
py_compile	Компиляция исходного программного кода на языке Python в файлы с байт-кодом
repr (reprlib)	Альтернативная реализация функции repr()
symbol	Константы, используемые для внутреннего представления узлов деревьев парсера
tabnanny	Определение некорректного оформления отступов
test	Пакет регрессивного тестирования
token	Конечные узлы дерева парсинга
tokenize	Сканер исходного программного кода на языке Python
user	Парсинг пользовательских файлов с настройками
zipimport	Импортирование модулей из zip-архивов

Обработка строк

Ниже приводится перечень некоторых ранее использовавшихся модулей, признанных в настоящее время устаревшими, которые предназначены для работы со строками.

Модуль	Описание
difflib	Вычисляет различия между строками
fpformat	Форматирование чисел с плавающей точкой
stringprep	Подготовка строк к передаче через Интернет
textwrap	Форматирование текста

Модули для доступа к службам операционной системы

Следующие модули обеспечивают доступ к дополнительным службам операционной системы. Некоторые функциональные возможности модулей, перечисленных здесь, уже включены в другие модули, описанные в главе 19 «Службы операционной системы».

Модуль	Описание
crypt	Доступ к функции crypt системы UNIX
curses	Интерфейс к библиотеке curses
grp	Доступ к базе данных с информацией о группах
pty	Работа с псевдотерминалами
pipes	Интерфейс к конвейеру командной оболочки
nis	Интерфейс к службе Sun NIS
platform	Доступ к информации о платформе
pwd	Доступ к базе данных с информацией о паролях
readline	Интерфейс к библиотеке GNU readline
rlcompleter	Функция автодополнения для библиотеки GNU readline
resource	Информация об использовании ресурсов
sched	Планировщик событий
spwd	Доступ к теневой базе данных с информацией о паролях
stat	Поддержка интерпретации результатов, возвращаемых функцией os.stat()
syslog	Интерфейс к демону syslog в UNIX
termios	Управление устройствами TTY в UNIX
tty	Функции управления терминалами

Сети

Следующие модули обеспечивают поддержку редко используемых сетевых протоколов:

Модуль	Описание
imaplib	Поддержка протокола IMAP
nntplib	Поддержка протокола NNTP
poplib	Поддержка протокола POP3
smtpd	Сервер SMTP
telnetlib	Поддержка протокола Telnet

Обработка и представление данных в Интернете

Следующие модули обеспечивают расширенную поддержку обработки и представления данных в Интернете, которые не были рассмотрены в главе 24 «Обработка и представление данных в Интернете».

Модуль	Описание
binhex	Поддержка формата BinHex4 файлов
formatter	Универсальные функции форматирования
mailcap	Обработка файлов в формате mailcap
mailbox	Чтение файлов почтовых ящиков в различных форматах
netrc	Обработка файлов в формате netrc
plistlib	Обработка файлов в формате Macintosh plist
uu	Поддержка формата uuencode файлов
xdrlib	Обработка данных в формате Sun XDR

Интернационализация

Следующие модули используются при разработке интернационализированных приложений:

Модуль	Описание
gettext	Средства обработки текста на нескольких языках
locale	Интерфейс к функциям интернационализации, предоставляемым системой

Мультимедийные службы

Следующие модули обеспечивают поддержку для работы с различными типами мультимедийных файлов:

Модуль	Описание
audioop	Средства для работы с «сырыми» аудиоданными
aifc	Чтение и запись файлов в форматах AIFF и AIFC
sunau	Чтение и запись файлов в формате Sun AU
wave	Чтение и запись файлов в формате WAV
chunk	Чтение блоков данных в формате IFF
colorsys	Преобразования между различными системами представления цвета
imghdr	Определение типов изображений
sndhdr	Определение типов аудиофайлов
ossaudiodev	Доступ к OSS-совместимым аудиоустройствам

Различные модули

В следующем заключительном списке перечислены модули, которые трудно отнести к какой-то определенной категории:

Модуль	Описание
cmd	Построчный интерпретатор команд
calendar	Функции создания календарей
shlex	Реализация простого лексического анализа
sched	Планировщик событий
Tkinter (tkinter)	Интерфейс к библиотеке Tcl/Tk для языка Python
winsound	Проигрывание звука в Windows

III

Расширение и встраивание

26

Расширение и встраивание интерпретатора Python

Одной из наиболее мощных особенностей языка Python является возможность организации взаимодействий интерпретатора с программами на языке C. Существует две основных стратегии интеграции Python с программным кодом, написанным на другом языке программирования. Первая: сторонние функции могут быть упакованы в модуль языка Python и импортированы с помощью инструкции `import`. Такие модули называют *модулями расширений*, потому что они расширяют возможности интерпретатора дополнительными функциями, написанными не на языке Python. Это наиболее распространенная форма интеграции Python-C, предоставляющая приложениям на языке Python доступ к высокопроизводительным библиотекам программного обеспечения. Другая форма интеграции Python-C – это *встраивание*, когда из программ на языке C осуществляется доступ к интерпретатору и к программам на языке Python, как к библиотекам. Этот подход иногда используется, когда по тем или иным причинам возникает необходимость встроить интерпретатор Python в существующий программный комплекс на языке C, обычно как некоторый механизм для работы со сценариями.

В этой главе рассматриваются базовые основы программного интерфейса Python-C. Сначала будут описаны основные составляющие программного интерфейса, встроенного в интерпретатор Python, используемого при создании модулей расширений на языке C. Этот раздел не может рассматриваться как учебное руководство, поэтому тем, кто впервые сталкивается с этой темой, рекомендуется предварительно познакомиться с документом «Extending and Embedding the Python Interpreter» (Расширение и встраивание интерпретатора Python), по адресу *http://docs.python.org/extending*, а также со справочным руководством «Python/C API Reference Manual» (Справочное руководство по прикладному программному интерфейсу Python/C), доступному по адресу *http://docs.python.org/c-api*. Затем будет описан библиотечный модуль `ctypes`. Это чрезвычайно полезный модуль, который можно использовать для организации доступа к функциям в биб-

лиотеках, написанных на языке C, без необходимости писать какой-либо программный код на языке C или использовать компилятор языка C.

Следует отметить, что при создании сложных расширений или приложений, куда внедряется интерпретатор Python, большинство программистов предпочитают обращаться к улучшенным генераторам программного кода и к библиотекам. В качестве примера можно привести проект SWIG (*http://www.swig.org*) – компилятор, который создает модули расширений Python на основе содержимого заголовочных файлов на языке C. Ссылки на этот и другие инструменты создания расширений можно найти по адресу *http://wiki.python.org/moin/IntegratingPythonWithOtherLanguages*.

Модули расширений

В этом разделе схематически описывается процедура создания собственного модуля расширения на языке C для Python. Создание модуля расширения заключается в построении интерфейса между интерпретатором Python и существующими функциями, написанными на языке C. Как правило, создание интерфейса к библиотеке на языке C начинается с создания заголовочного файла, такого, как показано ниже:

```
/* файл : example.h */
#include <stdio.h>
#include <string.h>
#include <math.h>

typedef struct Point {
    double x;
    double y;
} Point;

/* Поиск наибольшего общего делителя двух целых чисел x и y */
extern int gcd(int x, int y);

/* Замещает символ och на nch в строке s и возвращает количество замен */
extern int replace(char *s, char och, char nch);

/* Вычисляет расстояние между двумя точками */
extern double distance(Point *a, Point *b);

/* Константа препроцессора */
#define MAGIC 0x31337
```

Реализация этих функций находится в отдельном файле. Например:

```
/* example.c */
#include "example.h"
/* Поиск наибольшего общего делителя двух целых чисел x и y */
int gcd(int x, int y) {
    int g;
    g = y;
    while (x > 0) {
        g = x;
        x = y % x;
        y = g;
    }
```

```
    return g;
}

/* Замещает символ в строке */
int replace(char *s, char oldch, char newch) {
    int nrep = 0;
    while (s = strchr(s,oldch)) {
        *(s++) = newch;
        nrep++;
    }
    return nrep;
}

/* Расстояние между двумя точками */
double distance(Point *a, Point *b) {
    double dx,dy;
    dx = a->x - b->x;
    dy = a->y - b->y;
    return sqrt(dx*dx + dy*dy);
}
```

Ниже приводится основная программа на языке C с функцией `main()`, иллюстрирующая использование этих функций:

```
/* main.c */
#include "example.h"
int main() {
    /* Тест функции gcd() */
    {
        printf("%d\n", gcd(128,72));
        printf("%d\n", gcd(37,42));
    }
    /* Тест функции replace() */
    {
        char s[] = "Skipping along unaware of the unspeakable peril.";
        int nrep;
        nrep = replace(s,' ','-');
        printf("%d\n", nrep);
        printf("%s\n",s);
    }
    /* Тест функции distance() */
    {
        Point a = { 10.0, 15.0 };
        Point b = { 13.0, 11.0 };
        printf("%0.2f\n", distance(&a,&b));
    }
}
```

Ниже приводится вывод этой программы:

```
% a.out
8
1
6
Skipping-along-unaware-of-the-unspeakable-peril.
5.00
```

Прототип модуля расширения

Создание модуля расширения заключается в том, чтобы создать отдельный файл с программным кодом на языке C, содержащий набор функций-оберток, обеспечивающих связь между интерпретатором Python и программным кодом на языке C. Ниже приводится пример простого модуля расширения с именем _example:

```
/* pyexample.c */
#include "Python.h"
#include "example.h"

static char py_gcd_doc[] = "Поиск НОД двух целых чисел";
static PyObject *
py_gcd(PyObject *self, PyObject *args) {
    int x,y,r;
    if (!PyArg_ParseTuple(args,"ii:gcd",&x,&y)) {
        return NULL;
    }
    r = gcd(x,y);
    return Py_BuildValue("i",r);
}

static char py_replace_doc[] = "Замещает символы в строке";
static PyObject *
py_replace(PyObject *self, PyObject *args, PyObject *kwargs) {
    static char *argnames[] = {"s","och","nch",NULL};
    char     *s,*sdup;
    char      och, nch;
    int       nrep;
    PyObject *result;
    if (!PyArg_ParseTupleAndKeywords(args,kwargs, "scc:replace",
                                     argnames, &s, &och, &nch)) {
        return NULL;
    }
    sdup = (char *) malloc(strlen(s)+1);
    strcpy(sdup,s);
    nrep = replace(sdup,och,nch);
    result = Py_BuildValue("(is)",nrep,sdup);
    free(sdup);
    return result;
}

static char py_distance_doc[] = "Вычисляет расстояние между двумя точками";
static PyObject *
py_distance(PyObject *self, PyObject *args) {
    PyErr_SetString(PyExc_NotImplementedError,"distance() not implemented.");
    return NULL;
}

static PyMethodDef _examplemethods[] = {
    {"gcd", py_gcd, METH_VARARGS, py_gcd_doc},
    {"replace", py_replace, METH_VARARGS | METH_KEYWORDS, py_replace_doc},
    {"distance",py_distance,METH_VARARGS, py_distance_doc},
    {NULL, NULL, 0, NULL}
```

```
};

#if PY_MAJOR_VERSION < 3
/* Инициализация модуля в Python 2 */
void init_example(void) {
    PyObject *mod;
    mod = Py_InitModule("_example", _examplemethods);
    PyModule_AddIntMacro(mod,MAGIC);
}
#else
/* Инициализация модуля в Python 3 */
static struct PyModuleDef _examplemodule = {
    PyModuleDef_HEAD_INIT,
    "_example",    /* имя модуля */
    NULL,          /* строка документирования модуля, может быть NULL */
    -1,
    _examplemethods
};
PyMODINIT_FUNC
PyInit__example(void) {
    PyObject *mod;
    mod = PyModule_Create(&_examplemodule);
    PyModule_AddIntMacro(mod, MAGIC);
    return mod;
}
#endif
```

Модули расширений всегда должны подключать файл "Python.h". Для каждой функции на языке C, доступной из программного кода на языке Python, должна быть написана функция-обертка. Эти функции-обертки должны принимать два аргумента (`self` и `args`, оба типа `PyObject *`) или три (`self`, `args` и `kwargs`, все типа `PyObject *`). Аргумент `self` используется, когда функция-обертка реализует метод экземпляра некоторого класса. В этом случае экземпляр передается функции в аргументе `self`. В противном случае в аргументе `self` передается значение `NULL`. Аргумент `args` – кортеж с аргументами функции, которые передаются интерпретатором. Аргумент `kwargs` – словарь с именованными аргументами.

Преобразование значений аргументов из типов языка Python в типы языка C производится с помощью функции `PyArg_ParseTuple()` или `PyArg_ParseTupleAndKeywords()`. Таким же образом для преобразования возвращаемого значения используется функция `Py_BuildValue()`. Эти функции будут описаны в следующих разделах.

Строки документирования для функций в расширении должны помещаться в отдельные строковые переменные, например `py_gcd_doc` и `py_replace_doc`, как было показано выше. Эти переменные используются при инициализации модуля (о чем рассказывается чуть ниже).

Функции-обертки никогда не должны изменять данные, полученные от интерпретатора по ссылке, даже под страхом смерти. Именно по этой причине функция-обертка `py_replace()` создает копию полученной строки, прежде чем передать ее функции на языке C (которая модифицирует саму

строку). Если этот шаг пропустить, функция-обертка нарушит принцип неизменяемости строк, который действует в языке Python.

Если необходимо возбудить исключение, для этого следует вызвать функцию `PyExc_SetString()`, как это сделано в функции-обертке `py_distance()`. Возвращаемое значение `NULL` служит сигналом о возникшей ошибке.

Таблица методов `_examplemethods` используется для связи имен в языке Python с функциями-обертками на языке C. Эти имена используются для вызова функций из интерпретатора. Флаг `METH_VARARGS` указывает на то, какое соглашение о вызове применяется к функции-обертке. В данном случае функция-обертка принимает только позиционные аргументы в виде кортежа. В качестве значения флага может также передаваться комбинация `METH_VARARGS | METH_KEYWORDS`, которая указывает, что функция-обертка принимает также именованные аргументы. Дополнительно при создании таблицы методов устанавливаются строки документирования для каждой функции-обертки.

В заключительной части модуля расширения выполняется процедура инициализации, которая отличается для Python 2 и Python 3. В Python 2 для инициализации содержимого модуля используется функция инициализации `init_example`. В данном примере вызов функции `Py_InitModule("_example",_examplemethods)` создает модуль с именем `_example` и заполняет его объектами функций, соответствующих функциям, перечисленным в таблице методов. В Python 3 сначала необходимо создать объект `_examplemodule` класса `PyModuleDef`, описывающий модуль. Затем требуется написать функцию `PyInit__example()`, которая инициализирует модуль, как показано в примере. Кроме того, функция инициализации также является местом, где определяются значения констант и другие элементы модуля. Например, функция `PyModule_AddIntMacro()` добавляет в модуль значение препроцессора.

Следует отметить, что соглашение об именовании имеет критически важное значение для процедуры инициализации модуля. Если создается модуль с именем *modname*, в Python 2 функция инициализации модуля должна иметь имя `init`*modname*`()`, а в Python 3 – имя `PyInit_`*modname*`()`. Если не следовать этому соглашению, интерпретатор не сможет корректно загрузить модуль.

Выбор имен для модулей расширений

Стандартной практикой стало давать модулям расширений на языке C имена, начинающиеся с символа подчеркивания, такие как `'_example'`. Данное соглашение соблюдается и в самой стандартной библиотеке Python. Например, существуют модули `_socket`, `_thread`, `_sre` и `_fileio`, соответствующие программным компонентам на языке C модулей `socket`, `threading`, `re` и `io`. На практике модули расширений на языке C обычно не используются непосредственно. Для взаимодействия с ними создаются высокоуровневые модули на языке Python, например такие, как показано ниже:

```
# example.py
from _example import *
```

```
# Далее следует дополнительный программный код поддержки
...
```

Назначение такого модуля-обертки на языке Python состоит в том, чтобы обеспечить дополнительную поддержку для вашего модуля или предоставить более высокоуровневый интерфейс. Во многих случаях реализовать определенные части модуля расширения проще на языке Python, чем на C. Выбор такой архитектуры позволяет упростить решение этой задачи. Многие модули, входящие в состав стандартной библиотеки, представляют собой подобную смесь программного кода на языках C и Python.

Компилирование и упаковывание расширений

Предпочтительным механизмом компиляции и упаковки модулей расширений является пакет `distutils`. Чтобы воспользоваться им, необходимо создать файл `setup.py`, который выглядит, как показано ниже:

```
# setup.py
from distutils.core import setup, Extension

setup(name="example",
    version="1.0",
    py_modules = ['example.py'],
    ext_modules = [
        Extension("_example",
                  ["pyexample.c","example.c"])
    ]
)
```

В этом файле требовалось подключить файл с высокоуровневым модулем на языке Python (`example.py`) и исходные файлы, составляющие модуль расширения (`pyexample.c`, `example.c`). Чтобы собрать модуль для тестирования, нужно ввести команду:

```
% python setup.py build_ext --inplace
```

Она скомпилирует файлы с исходными текстами в разделяемую библиотеку и оставит ее в текущем рабочем каталоге. Эта библиотека получит имя `_examplemodule.so`, `_examplemodule.pyd` или другое аналогичное имя.

После успешной компиляции использование модуля выглядит очень просто. Например:

```
% python3.0
Python 3.0 (r30:67503, Dec 4 2008, 09:40:15)
[GCC 4.0.1 (Apple Inc. build 5465)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import example
>>> example.gcd(78,120)
6
>>> example.replace("Hello World",' ','-')
(1, 'Hello-World')
>>> example.distance()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
```

```
NotImplementedError: distance() not implemented.
(Перевод:
Трассировочная информация (самый последний вызов – самый нижний):
  Файл "<stdin>", строка 1, в <module>
NotImplementedError: функция distance() не реализована
)
>>>
```

Для сборки более сложных модулей расширения может потребоваться указать дополнительную информацию, например подключаемые каталоги и библиотеки и макроопределения препроцессора. Эту информацию также можно добавить в файл `setup.py`, как показано ниже:

```
# setup.py
from distutils.core import setup, Extension

setup(name="example",
    version="1.0",
    py_modules = ['example.py'],
    ext_modules = [
        Extension("_example",
                  ["pyexample.c","example.c"],
                  include_dirs = ["/usr/include/X11","/opt/include"],
                  define_macros = [('DEBUG','1'),
                                   ('MONDO_FLAG',1)],
                  undef_macros = ['HAVE_FOO','HAVE_NOT'],
                  library_dirs= ["/usr/lib/X11", "/opt/lib"],
                  libraries = [ "X11", "Xt", "blah" ])
    ]
)
```

Чтобы установить модуль для общего пользования, достаточно просто ввести команду `python setup.py install`. Дополнительные подробности по этой теме приводятся в главе 8 «Модули, пакеты и дистрибутивы».

В некоторых ситуациях может появиться необходимость собрать модуль расширения вручную. Для этого практически всегда требуется дополнительно иметь представление о различных параметрах компиляции и компоновки. Ниже приводится пример сборки модуля в Linux:

```
linux % gcc -c -fpic -I/usr/local/include/python2.6 example.c pyexample.c
linux % gcc -shared example.o pyexample.o -o _examplemodule.so
```

Преобразование типов данных языка Python в типы языка C

Ниже перечислены функции, которые используются модулями расширений для преобразования типов аргументов при передаче из программного кода на языке Python в программный код на языке C. Прототипы этих функций определены в заголовочном файле `Python.h`.

```
int PyArg_ParseTuple(PyObject *args, char *format, ...);
```

Преобразует кортеж *args* позиционных аргументов в последовательность переменных на языке C. В аргументе *format* передается строка формата,

содержащая ноль или более спецификаторов, которые приводятся в табл. 26.1–26.3, описывающих ожидаемое содержимое *args*. Все остальные аргументы содержат адреса переменных языка C, куда будут помещаться результаты преобразования. Порядок и типы аргументов должны соответствовать спецификаторам в строке *format*. Возвращает ноль, если не удалось преобразовать аргументы.

```
int PyArg_ParseTupleAndKeywords(PyObject *args, PyObject *kwargs, char *format,
char **kwlist, ...);
```

Преобразует кортеж *args* с позиционными аргументами и словарь *kwargs* с именованными аргументами. Аргумент *format* имеет тот же смысл, что и в функции `PyArg_ParseTuple()`. Единственное отличие состоит в том, что аргумент *kwlist* является списком строк, оканчивающимся нулевым символом и содержащим имена всех аргументов. В случае успеха возвращает `1`, в случае ошибки – `0`.

В **табл. 26.1 перечислены спецификаторы формата, которые можно использовать** в аргументе *format* для преобразования числовых значений. В колонке «Тип аргумента в языке C» указывается тип аргумента, который должен передаваться функциям `PyArg_Parse*()`. Для чисел – это всегда указатель на область в памяти, куда должен быть сохранен результат.

*Таблица 26.1. Преобразование чисел и соответствующие типы аргументов функций PyArg_Parse**

Формат	Тип в языке Python	Тип аргумента в языке C
"b"	Целое число	signed char *r
"B"	Целое число	unsigned char *r
"h"	Целое число	short *r
"H"	Целое число	unsigned short *r
"I"	Целое число	int *r
"I"	Целое число	unsigned int *r
"l"	Целое число	long int *r
"k"	Целое число	unsigned long *r
"L"	Целое число	long long *r
"K"	Целое число	unsigned long long *r
"n"	Целое число	Py_ssize_t *r
"f"	Число с плавающей точкой	float *r
"d"	Число с плавающей точкой	double *r
"D"	Комплексное число	Py_complex *r

Если при преобразовании целого числа со знаком значение, полученное из программного кода на языке Python, окажется слишком большим, чтобы уместиться в переменную с указанным типом языка C, будет возбуждено

исключение `OverflowError`. Однако при преобразованиях целых чисел без знака (таких как `'I'`, `'H'`, `'K'` и других) проверка на переполнение не выполняется и слишком большие значения будут просто усечены. При преобразовании в числа с плавающей точкой во входных аргументах могут передаваться значения типов `int` или `float` языка Python. В этом случае целые числа будут преобразованы в числа с плавающей точкой. В качестве чисел допускается передавать экземпляры пользовательских классов, при условии, что они предоставляют соответствующие методы преобразования, такие как `__int__()` или `__float__()`. Например, экземпляр класса, реализующий метод `__int__()`, может передаваться любой из перечисленных выше операций целочисленных преобразований (при этом метод `__int__()` будет вызван автоматически).

В табл. 26.2 перечислены спецификаторы формата, которые применяются к строкам и байтам. Многие операции преобразования возвращают в результате указатель и длину строки.

*Таблица 26.2. Преобразование строк и соответствующие типы аргументов функций PyArg_Parse**

Формат	Тип в языке Python	Тип аргумента в языке C
"c"	Строка или строка байтов с длиной 1	`char *r`
"s"	Строка	`char **r`
"s#"	Строка, строка байтов или буфер	`char **r, int *len`
"s*"	Строка, строка байтов или буфер	`Py_buffer *r`
"z"	Строка или `None`	`char **r`
"z#"	Строка, строка байтов или `None`	`char **r, int *len`
"z*"	Строка, строка байтов, буфер или `None`	`Py_buffer *r`
"y"	Строка байтов (оканчивающаяся нулевым символом)	`char **r`
"y#"	Строка байтов	`char **r, int *len`
"y*"	Строка байтов или буфер	`Py_buffer *r`
"u"	Строка (Юникода)	`Py_UNICODE **r`
"u#"	Строка (Юникода)	`Py_UNICODE **r, int *len`
"es"	Строка	`const char *enc, char **r`
"es#"	Строка или строка байтов	`const char *enc, char **r, int *len`
"et"	Строка или строка байтов (оканчивающаяся нулевым символом)	`const char *enc, char **r, int *len`
"et#"	Строка или строка байтов	`const char *enc, char **r, int *len`
"t#"	Буфер, доступный только для чтения	`char **r, int *len`

Формат	Тип в языке Python	Тип аргумента в языке C
"w"	Буфер, доступный для чтения/записи	char **r
"w#"	Буфер, доступный для чтения/записи	char **r, int *len
"w*"	Буфер, доступный для чтения/записи	Py_buffer *r

Обработка строк в расширениях на языке C представляет отдельную проблему, потому что тип данных char * используется для самых разных нужд. Например, указатель этого типа может ссылаться на текст, на одиночный символ или на буфер с двоичными данными. Еще одна проблема обусловлена тем, что в языке C для обозначения конца строки используется символ NULL ('\x00').

При передаче текста должны использоваться спецификаторы "s", "z", "u", "es" и "et", представленные в табл. 26.2. Эти спецификаторы не предполагают, что входной текст содержит завершающий символ NULL; в этом случае будет возбуждено исключение TypeError. Однако в программном коде на языке C можно смело полагаться на то, что строки будут завершаться символом NULL. В Python 2 допускается передавать как строки 8-битовых символов, так и строки Юникода, но в Python 3 все спецификаторы, за исключением "et", предполагают получить данные типа str языка Python и не могут использоваться для преобразования строк байтов. Когда строки Юникода передаются программному коду на языке C, они всегда кодируются с применением кодировки, используемой интерпретатором по умолчанию (обычно UTF-8). Единственное исключение составляет спецификатор "u", который возвращает строку Юникода во внутреннем представлении, используемом в языке Python. Это массив значений типа Py_UNICODE, где символы Юникода обычно представлены типом wchar_t языка C.

Спецификаторы "es" и "et" позволяют определить альтернативную кодировку текста. Для этих спецификаторов требуется указать название кодировки, такое как 'utf-8' или 'iso-8859-1', и текст будет преобразован в указанный формат и возвращен в виде буфера. Отличие спецификатора "et" от "es" заключается в том, что при передаче строки байтов предполагается, что строка уже была закодирована и передается дальше в исходном виде. Важно помнить, что при использовании спецификаторов "es" и "et" память для хранения результата выделяется динамически и ее необходимо явно освобождать вызовом функции PyMem_Free(). То есть программный код, использующий эти два вида преобразования, должен выглядеть примерно так:

```
PyObject *py_wrapper(PyObject *self, PyObject *args) {
    char *buffer;
    if (!PyArg_ParseTuple(args,"es","utf-8",&buffer)) {
        return NULL;
    }
    /* Выполнить какие-либо операции. */
    ...
```

```
    /* Освободить память и вернуть результат */
    PyMem_Free(buffer);
    return result;
}
```

Для обработки текста или двоичных данных используются спецификаторы "s#", "z#", "u#", "es#" или "et#". Эти спецификаторы действуют точно так же, как и описанные выше, за исключением того, что дополнительно они возвращают длину. Благодаря этому снимается ограничение на использование символа NULL. Кроме того, эти виды преобразований обеспечивают возможность работы со строками байтов и с любыми другими объектами, поддерживающими интерфейс буферов. *Интерфейс буфера* обеспечивает средства представления объектов в языке Python в виде простых двоичных буферов с содержимым этих объектов. Обычно этот интерфейс поддерживается строками, строками байтов и массивами (например, массивы, созданные средствами модуля array, поддерживают этот интерфейс). В этом случае, если объект предоставляет интерфейс буферов, доступных для чтения, возвращаются указатель на буфер и его размер. Наконец, если при использовании спецификаторов "es#" и "et#" переданы непустой указатель и ненулевая длина, предполагается, что они представляют буфер, уже размещенный в памяти, куда можно сохранить результат преобразования. В этом случае интерпретатор не будет выделять память для результата, и вам не потребуется вызывать функцию PyMem_Free().

Спецификаторы "s*" и "z*" по своему действию напоминают спецификаторы "s#" и "z#", но в отличие от последних они заполняют структуру Py_buffer информацией о принятых данных. Дополнительные сведения об этом можно найти в документе **PEP-3118, а здесь отмечу лишь, что эта структура** как минимум имеет атрибуты char *buf, int len и int itemsize, которые определяют адрес буфера в памяти, его размер (в байтах) и размер элементов, хранящихся в буфере. Кроме того, интерпретатор устанавливает блокировку на доступ к буферу, которая предотвращает его изменение другими потоками управления, пока буфер обрабатывается программным кодом расширения. Это позволяет расширению обрабатывать содержимое буфера независимо и даже, возможно, в потоке управления, отличном от того, в котором выполняется интерпретатор. В связи с этим на пользователя возлагается обязанность вызвать функцию PyBuffer_Release() после завершения работы с буфером.

Спецификаторы "t#", "w", "w#" и "w*" действуют точно так же, как семейство спецификаторов "s", за исключением того, что они принимают только объекты, реализующие интерфейс буферов. Спецификатор "t#" требует, чтобы буфер был доступен для чтения. Спецификатор "w" требует, чтобы буфер был доступен для чтения и для записи. Объект в языке **Python, поддерживающий** интерфейс буфера, доступного для записи, считается изменяемым объектом. Поэтому для таких объектов считается допустимым изменять их содержимое в расширениях на языке C.

Спецификаторы "y", "y#" и "y*" также напоминают семейство спецификаторов "s", за исключением того, что они принимают только строки байтов.

Эти спецификаторы должны использоваться только для обработки строк байтов и не способны работать со строками Юникода. Спецификатор "y" может принимать только строки байтов, не содержащие символы NULL.

В табл. 26.3 перечислены спецификаторы, которые принимают произвольные объекты языка Python и возвращают результат типа PyObject *. Они иногда используются в расширениях на языке C, которые должны обрабатывать объекты языка Python, более сложные, чем простые числа или строки, такие как экземпляры классов или словари.

*Таблица 26.3. Преобразование объектов языка Python и соответствующие типы аргументов функций PyArg_Parse**

Формат	Тип в языке Python	Тип аргумента в языке C
"O"	Любой	PyObject **r*
"O!"	Любой	PyTypeObject **type*, PyObject **r*
"O&"	Любой	int (**converter*)(PyObject *, void *), void **r*
"S"	Строка	PyObject **r*
"U"	Строка Юникода	PyObject **r*

Спецификаторы "O", "S" и "U" возвращают объекты Python типа PyObject *. Спецификаторы "S" и "U" ограничиваются объектами строк и строк Юникода соответственно.

Спецификатор "O!" требует наличия двух аргументов: указателя на объект типа языка Python и указателя PyObject * на указатель, который ссылается на местоположение объекта в памяти. Если тип объекта не соответствует объекту типа, возбуждается исключение TypeError. Например:

```
/* Преобразование списка аргументов */
PyObject *listobj;
PyArg_ParseTuple(args,"O!", &PyList_Type, &listobj);
```

В следующем списке перечислены имена типов языка C, соответствующие некоторым контейнерным типам в языке Python, которые часто участвуют в подобных преобразованиях.

Имя типа в языке C	Тип в языке Python
PyList_Type	list
PyDict_Type	dict
PySet_Type	set
PyFrozenSet_Type	frozen_set
PyTuple_Type	tuple
PySlice_Type	slice
PyByteArray_Type	bytearray

Спецификатор "O&" принимает два аргумента (*converter*, *addr*) и преобразует PyObject * в тип данных языка C. Аргумент *converter* – это указатель на функцию с сигнатурой `int converter(PyObject *obj, void *addr)`, где *obj* – это полученный объект на языке Python, а *addr* – адрес, который передается вторым аргументом функции PyArg_ParseTuple(). В случае успеха функция *converter*() должна возвращать 1, и 0 – в случае ошибки. Кроме того, в случае ошибки функция *converter*() должна возбудить исключение. Преобразования такого типа могут использоваться для отображения объектов языка Python, таких как списки или кортежи, в структуры данных на языке C. Например, ниже приводится одна из возможных реализаций функции-обертки py_distance() из ранее рассматривавшегося программного кода:

```
/* Преобразует кортеж в структуру Point. */
int convert_point(PyObject *obj, void *addr) {
    Point *p = (Point *) addr;
    return PyArg_ParseTuple(obj,"ii", &p->x, &p->y);
}
PyObject *py_distance(PyObject *self, PyObject *args) {
    Point p1, p2;
    double result;
    if (!PyArg_ParseTuple(args,"O&O&",
                          convert_point, &p1, convert_point, &p2)) {
        return NULL;
    }
    result = distance(&p1,&p2);
    return Py_BuildValue("d",result);
}
```

Наконец, аргумент *format* может содержать дополнительные модификаторы, имеющие отношение к распаковыванию кортежей, строкам документирования, сообщениям об ошибках и значениям аргументов по умолчанию. Эти модификаторы перечислены в следующем списке:

Формат	Описание
"(items)"	Описывает порядок распаковывания кортежей объектов. items содержит спецификаторы формата.
"|"	Начало списка необязательных аргументов.
":"	Конец списка аргументов. Остальной текст обозначает имя функции.
";"	Конец списка аргументов. Остальной текст – это сообщение об ошибке.

Модификатор "(items)" распаковывает значения, переданные в виде кортежа языка Python. Это может пригодиться для отображения кортежей в простые структуры на языке C. Например, ниже приводится еще одна возможная реализация функции-обертки py_distance():

```
PyObject *py_distance(PyObject *self, PyObject *args) {
    Point p1, p2;
    double result;
    if (!PyArg_ParseTuple(args,"(dd)(dd)",
                          &p1.x, &p1.y, &p2.x, &p2.y)) {
```

```
            return NULL;
        }
        result = distance(&p1,&p2);
        return Py_BuildValue("d",result);
    }
```

Модификатор "|" указывает, что все последующие аргументы являются необязательными. Этот модификатор может появляться в строке формата только один раз и не может быть вложенным. Модификатор ":" отмечает конец списка аргументов. Любой текст, следующий за ним, будет интерпретироваться, как имя функции для вывода в сообщениях об ошибках. Модификатор ";" также отмечает конец списка аргументов. Любой текст, следующий за ним, будет интерпретироваться, как текст сообщения об ошибке. Имейте в виду, что допускается использовать только какой-то один из модификаторов : и ;. Ниже приводятся несколько примеров:

```
PyArg_ParseTuple(args,"ii:gcd", &x, &y);
PyArg_ParseTuple(args,"ii; gcd requires 2 integers", &x, &y);

/* Преобразование с двумя необязательными аргументами */
PyArg_ParseTuple(args,"s|s", &buffer, &delimiter);
```

Преобразование типов данных языка C в типы языка Python

Для преобразования значений переменных языка **C в объекты языка Python** используется следующая функция:

```
PyObject *Py_BuildValue(char *format, ...)
```

Конструирует объект Python на основе последовательности переменных языка C. Аргумент *format* – это строка, описывающая требуемое преобразование. Остальные аргументы функции – значения переменных языка C, участвующие в преобразовании.

Спецификаторы формата, которые можно использовать в аргументе *format*, похожи на спецификаторы, используемые в функциях PyArg_ParseTuple*(), как видно из табл. 26.4.

Таблица 26.4. Спецификаторы формата для функции Py_BuildValue()

Формат	Тип в языке Python	Тип в языке C	Описание
""	None	void	Ничего.
"a"	Строка	char *	Строка, завершающаяся символом NULL. Если указатель на строку содержит значение NULL, возвращает значение None.
"s#"	Строка	char *, int	Строка и ее длина. Может содержать байты со значением '\x00'. Если указатель на строку содержит значение NULL, возвращает значение None.

Таблица 26.4 (продолжение)

Формат	Тип в языке Python	Тип в языке C	Описание
"y"	Строка байтов	char *	То же, что и "s", но возвращает строку байтов.
"y#"	Строка байтов	char *, int	То же, что и "s#", но возвращает строку байтов.
"z"	Строка или None	char *	То же, что и "s".
"z#"	Строка или None	char *, int	То же, что и "s#".
"u"	Строка Юникода	Py_UNICODE *	Строка Юникода, завершающаяся символом NULL. Если указатель на строку содержит значение NULL, возвращает значение None.
"u#"	Строка Юникода	Py_UNICODE *	Строка Юникода и ее длина.
"U"	Строка Юникода	char *	Преобразует строку языка C, завершающуюся символом NULL, в строку Юникода.
"U#"	Строка Юникода	char *, int	Преобразует строку языка C в строку Юникода.
"b"	Целое число	char	8-битное целое.
"B"	Целое число	unsigned char	8-битное целое без знака.
"h"	Целое число	short	16-битное короткое целое.
"H"	Целое число	unsigned short	16-битное короткое целое без знака.
"i"	Целое число	int	Целое число.
"I"	Целое число	unsigned int	Целое число без знака.
"l"	Целое число	long	Длинное целое число.
"L"	Целое число	unsigned long	Длинное целое число без знака.
"k"	Целое число	long long	Длинное целое типа long long.
"K"	Целое число	unsigned long long	Длинное целое типа long long без знака.
"n"	Целое число	Py_ssize_t	Тип size в языке Python.
"c"	Строка	char	Единственный символ. Создает строку языка Python с одним символом.
"f"	Число с плавающей точкой	float	Число с плавающей точкой одинарной точности.
"d"	Число с плавающей точкой	double	Число с плавающей точкой двойной точности.
"D"	Комплексное число	Py_complex	Комплексное число.

Формат	Тип в языке Python	Тип в языке C	Описание
"O"	Любой	PyObject *	Любой объект языка Python. Объект не изменяется, за исключением его счетчика ссылок, который увеличивается на 1. Если передается пустой указатель, возвращает также пустой указатель. Это удобно, когда ошибка возникла где-то в одном месте, а в другом необходимо продолжить ее распространение.
"O&"	Любой	функция преобразования, любой	Данные, полученные от программного кода на языке C, обрабатываются функцией преобразования.
"S"	Строка	PyObject *	То же, что и "O".
"N"	Любой	PyObject *	То же, что и "O", за исключением того, что счетчик ссылок объекта не увеличивается.
"(*items*)"	Кортеж	*vars*	Создает кортеж элементов *items*. *items* – строка со спецификаторами формата из этой таблицы. *vars* – список переменных языка C, соответствующих спецификаторам в *items*.
"[*items*]"	Список	*vars*	Создает список элементов *items*. *items* – строка со спецификаторами формата. *vars* – список переменных языка C, соответствующих спецификаторам в *items*.
"{*items*}"	Словарь	*vars*	Создает словарь элементов *items*.

Ниже приводятся несколько примеров создания значений различных типов:

```
Py_BuildValue("")                       None
Py_BuildValue("i",37)                   37
Py_BuildValue("ids",37,3.4,"hello")     (37, 3.5, "hello")
Py_BuildValue("s#","hello",4)           "hell"
Py_BuildValue("()")                     ()
Py_BuildValue("(i)",37)                 (37,)
Py_BuildValue("[ii]",1,2)               [1,2]
Py_BuildValue("[i,i]",1,2)              [1,2]
Py_BuildValue("{s:i,s:i}","x",1,"y",2)  {'x':1, 'y':2}
```

Строки Юникода создаются из данных типа `char *`; это означает, что исходные данные представляют собой последовательность байтов в кодировке по умолчанию (обычно UTF-8). Данные автоматически декодируют-

ся в строки Юникода при передаче программному коду на языке Python. Единственное исключение из этого правила составляют спецификаторы "y" и "y#", которые возвращают обычные строки байтов.

Добавление значений в модуль

В функции инициализации модуля расширения часто производится добавление констант и других значений. Для этой цели могут использоваться следующие функции:

`int PyModule_AddObject(PyObject *module, const char *name, PyObject *value)`

Добавляет в модуль новое значение. Аргумент *name* определяет имя значения, а аргумент *value* – объект Python, содержащий значение. Значения можно создавать с помощью функции `Py_BuildValue()`.

`int PyModule_AddIntConstant(PyObject *module, const char *name, long value)`

Добавляет в модуль целочисленное значение.

`void PyModule_AddStringConstant(PyObject *module, const char *name, const char *value)`

Добавляет в модуль строковое значение. В аргументе *value* должна передаваться строка, заканчивающаяся символом `NULL`.

`void PyModule_AddIntMacro(PyObject *module, macro)`

Добавляет в модуль целочисленное значение макроопределения. Аргумент *macro* должен быть именем макроопределения препроцессора.

`void PyModule_AddStringMacro(PyObject *module, macro)`

Добавляет в модуль строковое значение макроопределения.

Обработка ошибок

Модули расширений сообщают об ошибках, возвращая интерпретатору значение `NULL`. Перед тем как вернуть `NULL`, модуль должен определить исключение с помощью одной из следующих функций:

`void PyErr_NoMemory()`

Возбуждает исключение `MemoryError`.

`void PyErr_SetFromErrno(PyObject *exc)`

Возбуждает исключение *exc*. В аргументе *exc* передается объект исключения. Код ошибки для исключения извлекается из переменной `errno` в библиотеке языка C.

`void PyErr_SetFromErrnoWithFilename(PyObject *exc, char *filename)`

То же, что и `PyErr_SetFromErrno()`, но добавляет в исключение еще и имя файла.

`void PyErr_SetObject(PyObject *exc, PyObject *val)`

Возбуждает исключение *exc*. В аргументе *exc* передается объект исключения, а в аргументе *val* – объект со значениями атрибутов исключения.

```
void PyErr_SetString(PyObject *exc, char *msg)
```

Возбуждает исключение *exc*. В аргументе *exc* передается объект исключения, а в аргументе *msg* – сообщение, описывающее ошибку.

В аргументе *exc* допускается передавать одно из следующих значений:

Имя в языке C	Исключение в языке Python
PyExc_ArithmeticError	ArithmeticError
PyExc_AssertionError	AssertionError
PyExc_AttributeError	AttributeError
PyExc_EnvironmentError	EnvironmentError
PyExc_EOFError	EOFError
PyExc_Exception	Exception
PyExc_FloatingPointError	FloatingPointError
PyExc_ImportError	ImportError
PyExc_IndexError	IndexError
PyExc_IOError	IOError
PyExc_KeyError	KeyError
PyExc_KeyboardInterrupt	KeyboardInterrupt
PyExc_LookupError	LookupError
PyExc_MemoryError	MemoryError
PyExc_NameError	NameError
PyExc_NotImplementedError	NotImplementedError
PyExc_OSError	OSError
PyExc_OverflowError	OverflowError
PyExc_ReferenceError	ReferenceError
PyExc_RuntimeError	RuntimeError
PyExc_StandardError	StandardError
PyExc_StopIteration	StopIteration
PyExc_SyntaxError	SyntaxError
PyExc_SystemError	SystemError
PyExc_SystemExit	SystemExit
PyExc_TypeError	TypeError
PyExc_UnicodeError	UnicodeError
PyExc_UnicodeEncodeError	UnicodeEncodeError
PyExc_UnicodeDecodeError	UnicodeDecodeError
PyExc_UnicodeTranslateError	UnicodeTranslateError

(продолжение)

Имя в языке C	Исключение в языке Python
PyExc_ValueError	ValueError
PyExc_WindowsError	WindowsError
PyExc_ZeroDivisionError	ZeroDivisionError

Ниже перечислены функции, которые могут использоваться для получения от интерпретатора информации о наличии исключения или чтобы сбросить исключение:

`void PyErr_Clear()`

Сбрасывает ранее возбужденные исключения.

`PyObject *PyErr_Occurred()`

Проверяет, было ли возбуждено исключение. Если исключение было возбуждено, возвращает текущее значение исключения. В противном случае возвращает `NULL`.

`int PyErr_ExceptionMatches(PyObject *`*`exc`*`)`

Проверяет, соответствует ли текущее исключение исключению *exc*. Если соответствует, возвращает `1`, если не соответствует — `0`. При сопоставлении исключений эта функция применяет правила, определяемые языком Python. То есть *exc* может быть суперклассом текущего исключения или кортежем классов исключений.

Ниже приводится пример реализации блока `try-except` на языке C:

```
/* Выполнить некоторые операции над объектами языка Python */
if (PyErr_Occurred()) {
    if (PyErr_ExceptionMatches(PyExc_ValueError)) {
        /* выполнить действия по восстановлению после ошибки */
        ...
        PyErr_Clear();
        return result; /* Допустимый PyObject * */
    } else {
        return NULL;   /* Передать исключение интерпретатору */
    }
}
```

Подсчет ссылок

В отличие от программ, написанных на языке Python, **расширения на языке C имеют возможность манипулировать счетчиками ссылок объектов Python**. Делается это с помощью следующих макроопределений, каждое из которых применяется к объектам типа `PyObject *`.

Макроопределение	Описание
`Py_INCREF(`*`obj`*`)`	Увеличивает счетчик ссылок объекта, на который ссылается указатель *obj*. Указатель не должен быть пустым.

`Py_DECREF(obj)`	Уменьшает счетчик ссылок объекта, на который ссылается указатель *obj*. Указатель не должен быть пустым.
`Py_XINCREF(obj)`	Увеличивает счетчик ссылок объекта, на который ссылается указатель *obj*. Указатель может быть пустым.
`Py_XDECREF(obj)`	Уменьшает счетчик ссылок объекта, на который ссылается указатель *obj*. Указатель может быть пустым.

Манипулирование счетчиками ссылок объектов языка Python из программного кода на языке C – это достаточно тонкая тема, поэтому читателям настоятельно рекомендуется прочитать документ «Extending and Embedding the Python Interpreter» (Расширение и встраивание интерпретатора Python), по адресу *http://docs.python.org/extending*, прежде чем предпринимать что-либо. Как правило, в расширениях на языке C не приходится беспокоиться о счетчиках ссылок на объекты, за исключением следующих случаев:

- Если ссылка на объект Python сохраняется для последующего использования в структуре на языке C, счетчик ссылок необходимо увеличить.
- Аналогично при удалении ссылки, сохраненной ранее, счетчик ссылок необходимо уменьшить.
- При управлении контейнерными объектами языка Python (списками, словарями и другими) из программного кода на языке C может потребоваться реализовать управление счетчиками ссылок отдельных элементов. Например, высокоуровневые операции, которые извлекают или добавляют элементы в контейнер, обычно увеличивают счетчик ссылок.

Обычно ошибки при работе со счетчиками ссылок проявляются в аварийном завершении интерпретатора при попытке использовать расширение (вы забыли увеличить счетчик ссылок) или в виде утечек памяти при использовании функций из расширения (вы забыли уменьшить счетчик ссылок).

Потоки управления

Глобальная блокировка интерпретатора не позволяет интерпретатору выполнять более одного потока управления одновременно. Если функция, реализованная в модуле расширения, выполняется достаточно продолжительное время, она может заблокировать выполнение остальных потоков управления. Это обусловлено тем, что перед каждым вызовом функции в расширении приобретается блокировка. Если модуль расширения допускает возможность работы в многопоточном режиме, в нем можно воспользоваться следующими макроопределениями, позволяющими освобождать и повторно приобретать глобальную блокировку интерпретатора:

```
Py_BEGIN_ALLOW_THREADS
```

Освобождает глобальную блокировку интерпретатора и позволяет интерпретатору выполнять другие потоки управления одновременно с расширением. Расширение на языке C не должно вызывать функции Python C API после освобождения блокировки.

`Py_END_ALLOW_THREADS`

Повторно приобретает глобальную блокировку интерпретатора. Расширение будет приостановлено, пока блокировка не будет приобретена.

Следующий пример иллюстрирует использование этих макроопределений:

```
PyObject *py_wrapper(PyObject *self, PyObject *args) {
    ...
    PyArg_ParseTuple(args, ...)
    Py_BEGIN_ALLOW_THREADS
    result = run_long_calculation(args);
    Py_END_ALLOW_THREADS
    ...
    return Py_BuildValue(fmt,result);
}
```

Встраивание интерпретатора Python

Интерпретатор Python также может встраиваться в приложения на языке C. При встраивании интерпретатор Python действует, как библиотека, с помощью которой программы на языке C могут инициализировать интерпретатор, запускать с его помощью сценарии и фрагменты программного кода, загружать библиотечные модули, а также манипулировать функциями и объектами, реализованными на языке Python.

Порядок встраивания

При встраивании за запуск интерпретатора отвечает программа на языке C. Ниже приводится пример простой программы на языке C, которая иллюстрирует минимально возможный программный код, реализующий встраивание:

```
#include <Python.h>
int main(int argc, char **argv) {
    Py_Initialize();
    PyRun_SimpleString("print('Hello World')");
    Py_Finalize();
    return 0;
}
```

В этом примере программа инициализирует интерпретатор, выполняет короткий сценарий в виде строки и закрывает интерпретатор. Прежде чем углубляться в эту тему далее, обычно бывает полезно рассмотреть действующий пример.

Компиляция и связывание

Чтобы скомпилировать программу на языке C, которая встраивает интерпретатор, в системе UNIX, программа должна подключить заголовочный файл "Python.h" и быть скомпонована с библиотекой интерпретатора, такой как libpython2.6.a. Заголовочный файл обычно находится в каталоге /usr/local/include/python2.6, а библиотека – в каталоге /usr/local/lib/python2.6/

config. В системе Windows эти файлы находятся в каталоге установки Python. Имейте в виду, что интерпретатор может зависеть от других библиотек, которые вам также следует включить в инструкции компоновки. К сожалению, эта процедура зависит от типа платформы и от того, с какими параметрами была выполнена сборка интерпретатора Python, поэтому вам, возможно, придется повозиться.

Запуск и основные операции интерпретатора

Ниже перечислены функции, которые используются для запуска интерпретатора и выполнения сценариев с его помощью:

```
int PyRun_AnyFile(FILE *fp, char *filename)
```

Если *fp* представляет интерактивное устройство, например терминал в UNIX, эта функция вызовет функцию PyRun_InteractiveLoop(). В противном случае – функцию PyRun_SimpleFile(). В аргументе *filename* передается строка с именем входного потока. Это имя будет фигурировать в сообщениях интерпретатора об ошибках. Если в аргументе *filename* передать NULL, в качестве имени файла по умолчанию будет использоваться строка "???".

```
int PyRun_SimpleFile(FILE *fp, char *filename)
```

Похожа на функцию PyRun_SimpleString(), за исключением того, что программа на языке Python извлекается из файла *fp*.

```
int PyRun_SimpleString(char *command)
```

Выполняет команду *command* в контексте модуля __main__ интерпретатора. Возвращает 0 в случае успеха и -1 – в случае появления исключения.

```
int PyRun_InteractiveOne(FILE *fp, char *filename)
```

Выполняет единственную интерактивную команду.

```
int PyRun_InterativeLoop(FILE *fp, char *filename)
```

Запускает интерпретатор в интерактивном режиме.

```
void Py_Initialize()
```

Инициализирует интерпретатор Python. Эта функция должна вызываться перед любыми другими функциями C API интерпретатора, за исключением функций Py_SetProgramName(), PyEval_InitThreads(), PyEval_ReleaseLock() и PyEval_AcquireLock().

```
int Py_IsInitialized()
```

Возвращает 1, если интерпретатор был инициализирован, 0 – в противном случае.

```
void Py_Finalize()
```

Завершает работу интерпретатора, уничтожая все вложенные потоки управления и объекты, созданные после вызова Py_Initialize(). Обычно эта функция освобождает всю память, выделенную интерпретатором. Однако циклические ссылки и модули расширений способны вызвать утечки памяти, которые не могут быть устранены этой функцией.

`void Py_SetProgramName(char *name)`

Определяет имя программы, которое обычно можно найти в переменной `argv[0]` модуля `sys`. Эта функция должна вызываться только перед вызовом `Py_Initialize()`.

`char *Py_GetPrefix()`

Возвращает путь установки платформонезависимых файлов. Это то же самое значение, которое можно найти в переменной `sys.prefix`.

`char *Py_GetExecPrefix()`

Возвращает путь установки платформозависимых файлов. Это то же самое значение, которое можно найти в переменной `sys.exec_prefix`.

`char *Py_GetProgramFullPath()`

Возвращает полный путь к выполняемому файлу интерпретатора Python.

`char *Py_GetPath()`

Возвращает путь по умолчанию поиска модулей. Возвращаемая строка содержит имена каталогов, разделенные символом-разделителем, который используется текущей платформой (`:` – в UNIX, `;` – в DOS/Windows).

`int PySys_SetArgv(int argc, char **argv)`

Определяет параметры командной строки для заполнения значения переменной `sys.argv`. Эта функция должна вызываться до вызова функции `Py_Initialize()`.

Обращение к интерпретатору Python из программы на языке C

Несмотря на существование большого количества способов доступа к интерпретатору из программ на языке C, тем не менее чаще всего используются четыре основные операции:

- Импортирование библиотечных модулей Python (имитация инструкции `import`).
- Получение ссылок на объекты, определяемые в модулях.
- Вызов функций на языке Python, обращение к классам и методам.
- Обращение к атрибутам объектов (к данным или к методам)

Все эти операции могут выполняться с помощью следующих функций интерфейса C API доступа к программному коду Python:

`PyObject *PyImport_ImportModule(const char *modname)`

Импортирует модуль *modname* и возвращает ссылку на объект модуля.

`PyObject *PyObject_GetAttrString(PyObject *obj, const char *name)`

Возвращает значение атрибута объекта. Эквивалентно обращению к атрибуту *obj.name* в программе на языке Python.

`int PyObject_SetAttrString(PyObject *obj, const char *name, PyObject *value)`

Устанавливает значение атрибута объекта. Эквивалентно операции присваивания *obj.name = value* в программе на языке Python.

`PyObject *PyEval_CallObject(PyObject *func, PyObject *args)`

Вызывает функцию *func* с аргументами *args*. *func* – это вызываемый объект на языке Python (функция, метод, класс и так далее). *args* – кортеж аргументов.

`PyObject *PyEval_CallObjectWithKeywords(PyObject *func, PyObject *args, PyObject *kwargs)`

Вызывает функцию *func* с позиционными аргументами *args* и именованными аргументами *kwargs*. *func* – это вызываемый объект, *args* – кортеж и *kwargs* – словарь.

Следующий пример иллюстрирует использование этих функций для обращения к различным компонентам модуля `re` из программы на языке C. Эта программа выводит все строки, прочитанные из потока `stdin` и соответствующие регулярному выражению, переданному пользователем в виде аргумента.

```
#include "Python.h"
int main(int argc, char **argv) {
    PyObject *re;
    PyObject *re_compile;
    PyObject *pat;
    PyObject *pat_search;
    PyObject *args;
    char buffer[256];

    if (argc != 2) {
        fprintf(stderr,"Порядок использования: %s шаблон\n",argv[0]);
        exit(1);
    }

    Py_Initialize();

    /* import re */
    re = PyImport_ImportModule("re");

    /* pat = re.compile(pat,flags) */
    re_compile = PyObject_GetAttrString(re,"compile");
    args = Py_BuildValue("(s)", argv[1]);
    pat = PyEval_CallObject(re_compile, args);
    Py_DECREF(args);

    /* pat_search = pat.search - связанный метод */
    pat_search = PyObject_GetAttrString(pat,"search");

    /* Прочитать строки и выполнить сопоставление */
    while (fgets(buffer,255,stdin)) {
        PyObject *match;
        args = Py_BuildValue("(s)", buffer);
```

```
            /* match = pat.search(buffer) */
            match = PyEval_CallObject(pat_search,args);
            Py_DECREF(args);
            if (match != Py_None) {
                printf("%s",buffer);
            }
            Py_XDECREF(match);
        }
        Py_DECREF(pat);
        Py_DECREF(re_compile);
        Py_DECREF(re);
        Py_Finalize();
        return 0;
    }
```

При встраивании интерпретатора Python чрезвычайно важно обеспечить корректное управление счетчиками ссылок. В частности, необходимо уменьшать счетчики ссылок в любых объектах, созданных в программном коде на языке C **или возвращаемых программному коду на языке C в результате вызовов функций.**

Преобразование объектов на языке Python в объекты на языке C

Важной проблемой при использовании встроенного интерпретатора является преобразование результатов вызова функций или методов на языке Python в соответствующее представление на языке C. Как правило, при выполнении таких операций необходимо заранее точно знать тип данных, возвращаемых функцией. К сожалению, не существует высокоуровневой вспомогательной функции, такой как `PyArg_ParseTuple()`, для преобразования единственного объекта. Однако существует несколько низкоуровневых функций, которые перечислены ниже, способных преобразовывать некоторые простейшие типы данных **Python в соответствующее представление на языке C**, при условии, что заранее точно известно, преобразование какого объекта на языке Python выполняется:

Функции преобразования объектов Python в представление на языке C

`long`	`PyInt_AsLong(PyObject *)`
`long`	`PyLong_AsLong(PyObject *)`
`double`	`PyFloat_AsDouble(PyObject *)`
`char *`	`PyString_AsString(PyObject *)` **(только в Python 2)**
`char *`	`PyBytes_AsString(PyObject *)` **(только в Python 3)**

Если вам потребуется работать с более сложными типами данных, обращайтесь к описанию C API (*http://docs.python.org/c-api*).

Модуль ctypes

Модуль ctypes обеспечивает доступ к функциям в библиотеках DLL и в разделяемых библиотеках, написанных на языке C, из программного кода на языке Python. Хотя вам и потребуется знать определенные детали, относящиеся к библиотеке (имена, входные аргументы, типы аргументов и возвращаемых значений и так далее), вы можете использовать модуль ctypes для доступа к программному коду, написанному на языке C, без необходимости создавать функции-обертки и даже без необходимости использовать компилятор C. Модуль ctypes является важным модулем в стандартной библиотеке, обладающим весьма широкими функциональными возможностями. Ниже описываются значимые составляющие этого модуля, знакомство с которыми необходимо, чтобы начать работать с ним.

Загрузка разделяемых библиотек

Ниже перечислены классы, которые используются для загрузки разделяемых библиотек, написанных на языке C, и возвращают экземпляры, представляющие их содержимое:

CDLL(*name* [, *mode* [, *handle* [, *use_errno* [, *use_last_error*]]]])

Класс, представляющий стандартную разделяемую библиотеку языка C. В аргументе *name* передается имя библиотеки, такое как 'libc.so.6' или 'msvcrt.dll'. В аргументе *mode* передается флаг, который определяет, как будет загружаться библиотека и как она будет передаваться функции dlopen() в UNIX. Этот аргумент представляет собой битовую маску, составленную из флагов RTLD_LOCAL, RTLD_GLOBAL и RTLD_DEFAULT (по умолчанию), объединяемых битовой операцией ИЛИ. В Windows аргумент *mode* игнорируется. Аргумент *handle* определяет дескриптор уже загруженной библиотеки (если доступен). По умолчанию он принимает значение None. В аргументе *use_errno* передается логический флаг, добавляющий дополнительный уровень безопасности при работе с переменной errno языка C в загружаемой библиотеке. Когда задействован этот уровень, локальная копия переменной errno в текущем потоке управления сохраняется перед вызовом любых внешних функций и восстанавливается после вызова. По умолчанию аргумент *use_errno* принимает значение False. В аргументе *use_last_error* передается логический флаг, который разрешает использовать пару функций get_last_error() и set_last_error() для управления системным кодом ошибки. Чаще всего эти функции используются в Windows. По умолчанию аргумент *use_last_error* принимает значение False.

WinDLL(*name* [, *mode* [, *handle* [, *use_errno* [, *use_last_error*]]]])

То же, что и CDLL(), за исключением того, что в данном случае предполагается, что все функции в загружаемой библиотеке следуют стандартному соглашению Windows stdcall о вызовах (Windows).

Следующую функцию можно использовать, чтобы искать в системе разделяемые библиотеки и сконструировать имя, подходящее для использования в аргументе *name* предыдущих конструкторов классов. Эта функция определена в модуле ctypes.util:

`find_library(`*`name`*`)`

Определена в модуле `ctypes.util`. Возвращает полный путь к библиотеке с именем *`name`*. В аргументе *`name`* передается имя библиотеки без расширения, такое как `'libc'`, `'libm'` и так далее. Строка, возвращаемая функцией, содержит полный путь к файлу библиотеки, такой как `'/usr/lib/libc.so.6'`. Поведение этой функции в значительной степени зависит от типа платформы, от системных настроек размещения разделяемых библиотек и от окружения (например, от значения переменной окружения `LD_LIBRARY_PATH` и других параметров). Возвращает `None`, если библиотеку не удалось отыскать.

Внешние функции

Экземпляры разделяемых библиотек, созданные вызовом конструктора `CDLL()`, действуют как прокси-объекты, обеспечивающие доступ к содержимому библиотеки языка C. Чтобы обратиться к элементу библиотеки, достаточно просто использовать оператор доступа к атрибутам. Например:

```
>>> import ctypes
>>> libc = ctypes.CDLL("/usr/lib/libc.dylib")
>>> libc.rand()
16807
>>> libc.atoi("12345")
12345
>>>
```

В этом примере напрямую вызываются функции `libc.rand()` и `libc.atoi()`, имеющиеся в загруженной библиотеке на языке C.

Модуль `ctypes` предполагает, что все функции принимают аргумент типа `int` или `char *` и возвращают результат типа `int`. Поэтому, несмотря на то, что предыдущие вызовы были выполнены вполне успешно, вызовы других библиотечных функций на языке C могут работать не так, как ожидается. Например:

```
>>> libc.atof("34.5")
-1073746168
>>>
```

Чтобы устранить эту проблему, можно изменить параметры сигнатуры и особенности вызова любой внешней функции *`func`* с помощью следующих атрибутов:

`func``.argtypes`

Кортеж типов данных, определенных в модуле `ctypes` (рассказывается ниже) и описывающих входные аргументы функции *`func`*.

`func``.restype`

Тип данных, определенный в модуле `ctypes`, описывающий возвращаемое значение функции *`func`*. Для функций, возвращающих результат типа `void`, используется значение `None`.

func.errcheck

Вызываемый объект языка Python, принимающий три аргумента (*result*, *func*, *args*), где *result* – это значение, возвращаемое внешней функцией, *func* – ссылка на внешнюю функцию и *args* – кортеж входных аргументов. Эта функция вызывается после вызова внешней функции и может использоваться для проверки наличия ошибок и выполнения других действий.

Ниже приводится пример исправления ошибки в вызове функции atof(), показанной в предыдущем примере:

```
>>> libc.atof.restype=ctypes.c_double
>>> libc.atof("34.5")
34.5
>>>
```

ctypes.d_double – это ссылка на предопределенный тип данных. Этот и другие типы данных описываются в следующем разделе.

Типы данных

В табл. 26.5 перечислены типы данных, объявленные в модуле ctypes, которые могут использоваться в качестве значений атрибутов argtypes и restype внешних функций. Колонка «Значение в языке Python» описывает тип данных языка Python, который соответствует указанному типу данных.

Таблица 26.5. Типы данных в модуле ctypes

Имя типа в модуле ctypes	Тип данных в языке C	Значение в языке Python
c_bool	bool	True или False
c_bytes	signed char	Короткое целое число
c_char	char	Одиночный символ
c_char_p	char *	Строка или строка байтов, завершающаяся символом NULL
c_double	double	Число с плавающей точкой
c_longdouble	long double	Число с плавающей точкой
c_float	float	Число с плавающей точкой
c_int	int	Целое число
c_int8	signed char	8-битное целое число
c_int16	short	16-битное целое число
c_int32	int	32-битное целое число
c_int64	long long	64-битное целое число
c_long	long	Целое число
c_longlong	long long	Целое число
c_short	short	Целое число

Таблица 26.5 (продолжение)

Имя типа в модуле ctypes	Тип данных в языке C	Значение в языке Python
c_size_t	size_t	Целое число
c_ubyte	unsigned char	Целое число без знака
c_uint	unsigned int	Целое число без знака
c_uint8	unsigned char	8-битное целое число без знака
c_uint16	unsigned short	16-битное целое число без знака
c_uint32	unsigned int	32-битное целое число без знака
c_uint64	unsigned long long	64-битное целое число без знака
c_ulong	unsigned long	Целое число без знака
c_ulonglong	unsigned long long	Целое число без знака
c_ushort	unsigned short	Целое число без знака
c_void_p	void *	Целое число
c_wchar	wchar_t	Одиночный символ Юникода
c_wchar_p	wchar_t *	Строка Юникода, завершающаяся символом NULL

Чтобы создать тип, представляющий указатель языка C, к одному из перечисленных типов данных следует применить следующую функцию:

`POINTER(`*type*`)`

Определяет тип, который является указателем на значение типа *type*. Например, вызов `POINTER(c_int)` представляет тип данных языка C `int *`.

Чтобы определить тип, представляющий массив фиксированного размера, достаточно умножить существующий тип на количество элементов в массиве. Например, выражение `c_int*4` представляет тип данных `int[4]` в языке C.

Чтобы определить тип данных, представляющий структуру или объединение в языке C, необходимо создать класс, производный от одного из базовых классов `Structure` или `Union`. Внутри определения производного класса необходимо объявить атрибут класса `_fields_`, описывающий его содержимое. `_fields_` – это список кортежей из 2 или 3 элементов вида (*name*, *ctype*) или (*name*, *ctype*, *width*), где поле *name* – это идентификатор поля структуры, *ctype* – класс, описывающий тип поля, и *width* – целое число, определяющее ширину битового поля. Например, взгляните на следующую структуру на языке C:

```
struct Point {
    double x, y;
};
```

Описание этой структуры в терминах модуля `ctypes` имеет следующий вид:

```
class Point(Structure):
    _fields_ = [ ("x", c_double),
                 ("y", c_double) ]
```

Вызов внешних функций

Чтобы вызвать функцию из библиотеки, достаточно просто обратиться к соответствующему методу с набором аргументов, типы которых совместимы с сигнатурой. Для простых типов данных, таких как `c_int`, `c_double` и других, можно просто передавать значения соответствующих им типов в языке Python (целые числа, числа с плавающей точкой и другие). Допускается также передавать экземпляры типов `c_int`, `c_double` и подобных им. В случае массивов допускается передавать последовательности языка Python совместимых типов.

Чтобы передать внешней функции указатель, необходимо сначала создать экземпляр типа, представляющий значение, на которое будет ссылаться указатель, а затем создать объект указателя с помощью одной из следующих функций:

`byref(`*`cvalue`* `[,` *`offset`*`])`

Представляет легковесный указатель на значение *cvalue*. Аргумент *cvalue* должен быть экземпляром типа данных, определяемого модулем `ctypes`. Аргумент *offset* определяет смещение в байтах, которое следует прибавить к значению указателя. Значение, возвращаемое этой функцией, можно использовать только в вызовах внешних функций.

`pointer(`*`cvalue`*`)`

Создает экземпляр указателя, ссылающегося на значение *cvalue*. Аргумент *cvalue* должен быть экземпляром типа данных, определяемого модулем `ctypes`. Эта функция создает экземпляр типа `POINTER`, описанного выше.

Ниже приводится пример, демонстрирующий, как функции на языке C можно передать аргумент типа `double *`:

```
dval = c_double(0.0)   # Создать экземпляр типа double
r = foo(byref(dval))   # Вызвать foo(&dval)

p_dval = pointer(dval) # Создать переменную-указатель
r = foo(p_dval)        # Вызвать foo(p_dval)

# Проверить значение dval после вызова функции
print (dval.value)
```

Следует отметить, что нет никакой возможности создать указатель на значение встроенного типа, такого как `int` или `float`. Передача указателей на значения таких типов противоречила бы принципу неизменяемости, если внутри функции на языке C выполнялась бы попытка изменить значение.

Атрибут *cobj*`.value` экземпляра типа *cobj*, созданного средствами модуля `ctypes`, содержит само значение. Например, в предыдущем фрагменте при обращении к атрибуту `dval.value` возвращается значение с плавающей точкой, хранящееся в экземпляре `dval` класса `c_double`.

Чтобы передать функции на языке C структуру, следует создать экземпляр структуры или объединения. Для этого требуется вызвать конструктор ранее объявленного класса структуры или объединения *StructureType*:

StructureType(**args, **kwargs*)

Создает экземпляр *StructureType*, где под *StructureType* подразумевается класс, производный от класса `Structure` или `Union`. Позиционные аргументы в **args* используются для инициализации полей структуры; они должны следовать в том же порядке, в каком они перечислены в атрибуте `_fields_`. Именованные аргументы в ***kwargs* используются для инициализации только именованных полей структуры.

Альтернативные методы конструирования типов

Все экземпляры типов данных модуля `ctypes`, такие как `c_int`, `POINTER` и другие, обладают методами класса, которые могут использоваться для создания экземпляров типов данных `ctypes` из других объектов, размещаемых в памяти.

ty.`from_buffer`(*source* [,*offset*])

Создает экземпляр типа *ty*, использующий тот же буфер в памяти, что и объект *source*. Аргумент *source* должен быть объектом любого типа, который поддерживает интерфейс буферов, доступных для записи (например, `bytearray`, массив объектов `array`, определяемый модулем `array`, `mmap` и так далее). Аргумент *offset* определяет смещение используемой области в байтах относительно начала буфера.

ty.`from_buffer_copy`(*source* [, *offset*])

То же, что и *ty*.`from_buffer()`, за исключением того, что создает копию буфера, благодаря чему объект *source* может быть доступен только для чтения.

ty.`from_address`(*address*)

Создает экземпляр типа *ty*, извлекая значение из области памяти с адресом *address*, который определяется, как целое число.

ty.`from_param`(*obj*)

Создает экземпляр типа *ty* из объекта *obj* на языке Python. Этот метод может принимать только объекты *obj*, которые могут быть приведены к соответствующему типу. Например, целое число в языке Python может быть преобразовано в экземпляр типа `c_int`.

ty.`in_dll`(*library*, *name*)

Создает экземпляр типа *ty* из переменной в разделяемой библиотеке. В аргументе *library* передается экземпляр загруженной библиотеки, например объект, созданный конструктором класса `CDLL`. В аргументе *name* передается имя переменной. Этот метод можно использовать для создания оберток вокруг глобальных переменных, объявленных в библиотеке.

Следующий пример демонстрирует, как можно создать ссылку на глобальную переменную `int status`, объявленную в библиотеке `libexample.so`.

```
libexample = ctypes.CDLL("libexample.so")
status = ctypes.c_int.in_dll(libexample,"status")
```

Вспомогательные функции

Ниже перечислены вспомогательные функции, объявленные в модуле `ctypes`:

`addressof(`*`cobj`*`)`

Возвращает адрес объекта *`cobj`* в памяти в виде целого числа. Аргумент *`cobj`* должен быть экземпляром типа из модуля `ctypes`.

`alignment(`*`ctype_or_obj`*`)`

Возвращает целочисленную величину выравнивания для типа, объявленного в модуле `ctypes`, или объекта. В аргументе *`ctype_or_obj`* передается один из типов модуля `ctypes` или экземпляр такого типа.

`cast(`*`cobj`*`, `*`ctype`*`)`

Приводит объект *`cobj`* типа, объявленного в модуле `ctypes`, к новому типу *`ctype`*. Этот метод работает только с указателями, поэтому аргумент *`cobj`* должен быть указателем или массивом, а аргумент *`ctype`* должен быть типом указателя.

`create_string_buffer(`*`init`* `[, `*`size`*`])`

Создает символьный буфер, доступный для записи, в виде массива элементов типа `c_char`. В аргументе *`init`* может передаваться целое число, определяющее размер буфера, или строка, определяющая начальное содержимое буфера. *`size`* — это необязательный аргумент, который определяет размер буфера, когда в аргументе *`init`* передается строка. По умолчанию аргумент *`size`* получает значение на единицу больше, чем количество символов в *`init`*. Строки Юникода преобразуются в строки байтов с использованием кодировки по умолчанию.

`create_unicode_buffer(`*`init`* `[, `*`size`*`])`

То же, что и `create_string_buffer()`, за исключением того, что создает массив элементов типа `c_wchar`.

`get_errno()`

Возвращает текущее значение частной, для модуля `ctypes`, копии переменной `errno`.

`get_last_error()`

Возвращает текущее значение частной, для модуля `ctypes`, копии переменной `LastError` в Windows.

`memmove(`*`dst`*`, `*`src`*`, `*`count`*`)`

Копирует *`count`* байтов из *`src`* в *`dst`*. В аргументах *`src`* и *`dst`* допускается передавать целые числа, представляющие адреса значений в памяти или экземпляры типов, объявленных в модуле `ctypes`, которые можно преобразовать в указатели. Действует точно так же, как и функция `memmove()` в стандартной библиотеке языка C.

`memset(dst, c, count)`

Записывает *count* байтов со значением *c* в область памяти, начиная с адреса *dst*. В аргументе *dst* допускается передавать целые числа или экземпляры типов, объявленных в модуле `ctypes`. В аргументе *c* передается целое число в диапазоне 0-255, представляющее значение байта.

`resize(cobj, size)`

Изменяет размер внутренней памяти, используемой для представления объекта *cobj* одного из типов модуля `ctypes`. Аргумент *size* определяет новый размер в байтах.

`set_conversion_mode(encoding, errors)`

Определяет кодировку, которая будет использоваться при преобразовании строк Юникода в строки 8-битовых символов. Аргумент *encoding* определяет название кодировки, такое как `'utf-8'`, а аргумент *errors* – политику обработки ошибок кодирования, например: `'strict'` или `'ignore'`. Возвращает кортеж (*encoding, errors*) с предыдущими значениями настроек.

`set_errno(value)`

Записывает значение в частную, для модуля `ctypes`, копию системной переменной `errno`. Возвращает предыдущее значение.

`set_last_error(value)`

Записывает значение в частную, для модуля `ctypes`, копию переменной `LastError` в системе Windows и возвращает предыдущее значение.

`sizeof(type_or_cobj)`

Возвращает размер типа, объявленного в модуле `ctypes`, или объекта в байтах.

`string_at(address [, size])`

Возвращает строку байтов, представляющую *size* байтов в памяти, начиная с адреса *address*. Если аргумент *size* опущен, предполагается, что строка байтов заканчивается символом `NULL`.

`wstring_at(address [, size])`

Возвращает строку Юникода, представляющую *size* многобайтовых символов, начиная с адреса *address*. Если аргумент *size* опущен, предполагается, что строка байтов заканчивается символом `NULL`.

Пример

Следующий пример иллюстрирует применение модуля `ctypes` для построения интерфейса к набору функций на языке C, использовавшихся в самой первой части этой главы, где описывались особенности создания модулей расширений Python вручную.

```
# example.py

import ctypes
_example = ctypes.CDLL("./libexample.so")
```

```
# int gcd(int, int)
gcd = _example.gcd
gcd.argtypes = (ctypes.c_int,
                ctypes.c_int)
gcd.restype = ctypes.c_int

# int replace(char *s, char olcdh, char newch)
_example.replace.argtypes = (ctypes.c_char_p,
                             ctypes.c_char,
                             ctypes.c_char)
_example.replace.restype = ctypes.c_int

def replace(s, oldch, newch):
    sbuffer = ctypes.create_string_buffer(s)
    nrep = _example.replace(sbuffer,oldch,newch)
    return (nrep,sbuffer.value)

# double distance(Point *p1, Point *p2)
class Point(ctypes.Structure):
    _fields_ = [ ("x", ctypes.c_double),
                 ("y", ctypes.c_double) ]

_example.distance.argtypes = (ctypes.POINTER(Point),
                              ctypes.POINTER(Point))
_example.distance.restype = ctypes.c_double

def distance(a,b):
    p1 = Point(*a)
    p2 = Point(*b)
    return _example.distance(byref(p1),byref(p2))
```

В общем случае использование модуля `ctypes` всегда связано с созданием промежуточного уровня на языке той или иной сложности. Например, некоторые функции на языке C могут вызываться непосредственно. Однако иногда может потребоваться реализовать промежуточный уровень, чтобы учесть некоторые особенности программного кода на языке C. Так, как было показано в этом примере, чтобы вызвать функцию `replace()`, необходимо выполнить дополнительные действия, чтобы учесть тот факт, что функция из библиотеки на языке C изменяет содержимое входного буфера. Для вызова функции `distance()` потребовалось выполнить дополнительные операции, чтобы создать из кортежей экземпляры класса `Point` и передать функции указатели на них.

Примечание

Модуль `ctypes` обладает огромным количеством дополнительных особенностей, которые не рассматривались здесь. Например, модуль позволяет обращаться к самым разным библиотекам в системе Windows и обеспечивает поддержку функций обратного вызова, неполных типов и других особенностей. В электронной документации можно найти массу примеров, благодаря чему она может служить отличной отправной точкой к дальнейшему изучению модуля.

Дополнительные возможности расширения и встраивания

Создание модулей расширений вручную или использование модуля ctypes обычно не вызывает сложностей, если программный код расширения на языке C достаточно прост. Однако с ростом сложности расширения процесс его создания быстро может стать весьма трудоемким. Поэтому у вас может появиться желание отыскать подходящий инструмент создания расширений. Подобные инструменты либо автоматизируют большую часть процесса создания расширения, либо предоставляют программный интерфейс, позволяющий работать на более высоком уровне. Ссылки на различные инструменты можно найти на странице *http://wiki.python.org/moin/IntegratingPythonWithOtherLanguages*. Ниже приводится короткий пример использования инструмента SWIG (*http://www.swig.org*), исключительно в демонстрационных целях. Для полноты информации следует отметить, что первоначально инструмент SWIG был создан автором книги.

Используя инструменты автоматизации, программист обычно лишь создает описание содержимого модуля расширения на верхнем уровне. Например, при использовании SWIG создается лишь краткая спецификация интерфейса, которая выглядит, как показано ниже:

```
/* example.i : Пример спецификации Swig */
%module example
%{
/* Преамбула. Здесь выполняется подключение всех заголовочных файлов */
#include "example.h"
%}
/* Содержимое модуля. Здесь перечисляются все объявления на языке C */
typedef struct Point {
    double x;
    double y;
} Point;
extern int     gcd(int, int);
extern int     replace(char *s, char oldch, char newch);
extern double  distance(Point *a, Point *b);
```

Используя такую спецификацию, SWIG автоматически генерирует все, что должно содержаться в модуле расширения на языке Python. Чтобы запустить SWIG, достаточно просто вызвать его как компилятор:

```
% swig -python example.i
%
```

В результате создается набор файлов с расширениями .c и .py. При этом вам очень редко придется беспокоиться об их содержимом. Если вы пользуетесь пакетом distutils, включите файл спецификации с расширением .i в параметры настройки в файле setup.py; это обеспечит автоматический запуск SWIG при сборке расширения. Например, ниже приводится пример содержимого файла setup.py, который, благодаря включению файла example.i, автоматически запускает SWIG.

```
# setup.py
from distutils.core import setup, Extension
```

```
setup(name="example",
      version="1.0",
      py_modules = ['example.py'],
      ext_modules = [
          Extension("_example",
                    ["example.i","example.c"])
      ]
)
```

Оказывается, что такого файла `example.i` и файла `setup.py` вполне достаточно, чтобы создать действующий модуль расширения. Если выполнить команду `python setup.py build_ext --inplace`, она создаст полностью работоспособное расширение в каталоге.

Jython и IronPython

Возможности расширения и встраивания не ограничиваются программами на языке C. Тем, кто работает с языком Java, можно порекомендовать обратить внимание на Jython (*http://www.jython.org*) – полноценную реализацию интерпретатора Python на языке Java. Интерпретатор `jython` позволяет легко и просто импортировать библиотеки Java с помощью привычной инструкции `import`. Например:

```
bash-3.2$ jython
Jython 2.2.1 on java1.5.0_16
Type "copyright", "credits" or "license" for more information.
>>> from java.lang import System
>>> System.out.println("Hello World")
Hello World
>>>
```

Тем, кто работает с фреймворком .NET в Windows, можно порекомендовать обратить внимание на IronPython (*http://www.codeplex.com/Wiki/View.aspx?ProjectName=IronPython*) – полноценную реализацию интерпретатора Python на языке C#. Интерпретатор IronPython обеспечивает простой доступ к любым библиотекам .NET из программного кода на языке Python. Например:

```
% ipy
IronPython 1.1.2 (1.1.2) on .NET 2.0.50727.42
Copyright (c) Microsoft Corporation. All rights reserved.
>>> import System.Math
>>> dir(System.Math)
['Abs', 'Acos', 'Asin', 'Atan', 'Atan2', 'BigMul', 'Ceiling', 'Cos', 'Cosh',
...]
>>> System.Math.Cos(3)
-0.9899924966
>>>
```

Подробное рассмотрение интерпретаторов Jython и IronPython выходит далеко за рамки этой книги. Тем не менее имейте в виду, что оба они являются интерпретаторами языка Python, главное различие между которыми заключается в их библиотеках.

A

Python 3

В декабре 2008 года вышла версия Python 3.0. Изменения в языке Python оказались настолько существенными, что была нарушена обратная совместимость с Python 2 во многих важных областях. Достаточно полный обзор изменений в версии Python 3 можно найти в документе «What's New in Python 3.0» (Что нового в Python 3.0), доступном по адресу *http://docs.python.org/3.0/whatsnew/3.0.html*[1]. В некотором смысле, первые 26 глав этой книги можно рассматривать, как диаметральную противоположность документу «What's New». Основное внимание в этих главах уделялось особенностям, которые являются общими для Python 2 и Python 3, включая модули стандартной библиотеки, основные возможности языка и примеры. Кроме нескольких незначительных различий в именах и того факта, что инструкция `print()` стала функцией, никаких уникальных особенностей Python 3 в этой книге не рассматривалось.

Основная цель этого приложения состоит в том, чтобы дать описание новых возможностей языка Python, доступных только в версии 3, а также познакомить с некоторыми важными отличиями тех, кто планирует перевести существующий программный код на использование новой версии. В конце этого приложения приводятся несколько стратегий перехода и описывается утилита `2to3`, позволяющая упростить такой переход.

Кто должен использовать Python 3?

Прежде чем двинуться дальше, важно ответить на вопрос: «Кто должен использовать версию Python 3.0?». В сообществе пользователей Python всегда было известно, что переход на использование Python 3 будет происходить постепенно и что ветка Python 2 будет поддерживаться еще в течение некоторого времени (речь идет о годах). К моменту написания этих строк не было никакой срочной необходимости отказываться от программного кода, написанного для версии Python 2. Я полагаю, что даже спустя не-

[1] Аналогичную информацию (хотя это и не прямой перевод указанного документа) на русском языке можно найти на странице *http://www.ibm.com/developerworks/ru/library/l-python3-1/*. – *Прим. перев.*

сколько лет, когда увидит свет пятое издание этой книги, в разработке все еще будет находиться огромный объем программного кода для Python 2.

Главная проблема, которая возникает при переходе на Python 3.0, связана с совместимостью со сторонними библиотеками. Мощь языка **Python в значительной степени обусловлена** огромным разнообразием фреймворков и библиотек. Однако без явного вмешательства в реализацию этих библиотек они в подавляющем большинстве не способны работать под управлением Python 3. Эта проблема осложняется еще и тем, что многие библиотеки зависят от других библиотек, которые в свою очередь зависят от еще большего количества библиотек. К моменту написания этих строк (2009 год) еще существовали библиотеки и фреймворки для языка Python, которые не были даже перенесены на версию Python 2.4, не говоря уже о версии 2.6 или 3.0. Поэтому, если вы предполагаете использовать сторонний программный код, лучше пока остановиться на версии **Python 2. Если вы читаете эту книгу в 2012 году, можно надеяться, что ситуация значительно улучшилась.**

Несмотря на то что в версии **Python 3 были ликвидированы многие недостатки языка, тем не менее пока еще трудно сказать, что Python 3 является** достаточно обоснованным выбором для начинающих пользователей, которые только пытаются освоить основы языка. Практически вся документация, руководства, сборники рецептов и примеров написаны в предположении использования Python 2 и используют стиль программирования, несовместимый с Python 3. Разумеется, трудно получить положительный результат при обучении, если практически все примеры оказываются неработоспособными. Даже официальная документация до сих пор не полностью учитывает особенности программирования для версии Python 3; в процессе работы над книгой автор отправил огромное количество отчетов об ошибках, касающихся неточностей и упущений в документации.

Наконец, хотя Python 3.0 подается, как самая последняя и самая лучшая версия, тем не менее в ней немало проблем, связанных с производительностью и особенностями поведения. Например, система ввода-вывода в первых выпусках страдала недопустимо низкой производительностью. Разделение строк на строки байтов и строки Юникода также принесло свои проблемы. Даже некоторые встроенные модули оказались неработоспособными из-за изменений в системе ввода-вывода и в реализации механизма работы со строками. Очевидно, что с течением времени эти проблемы будут сняты, – по мере того, как все больше программистов будут участвовать в тестировании новых версий. Однако, по мнению автора этой книги, версия **Python 3.0 пока пригодна лишь для экспериментального использования** опытными ветеранами Python. Если вам требуется стабильность и высокая надежность, продолжайте пользоваться Python 2, пока в ветке Python 3 не будут устранены основные неувязки.

Новые возможности языка

В этом разделе рассматриваются некоторые новые возможности Python 3, которые *не* поддерживаются в Python 2.

Кодировка исходных текстов и идентификаторы

В Python 3 предполагается, что исходные тексты программ набираются в кодировке UTF-8. Кроме того, были ослаблены требования к набору символов, которые могут использоваться в идентификаторах. В частности, идентификаторы могут включать любые допустимые символы Юникода с кодовыми пунктами от U+0080 и выше. Например:

```
π = 3.141592654
r = 4.0
print(2*π*r)
```

Однако наличие возможности использовать такие символы в исходных текстах еще не означает, что ею следует бездумно пользоваться. Не все текстовые редакторы, терминалы и инструменты разработчика одинаково хорошо обрабатывают символы Юникода. Кроме того, для программистов может оказаться чрезвычайно неудобным нажимать неуклюжие комбинации клавиш, чтобы ввести символы, отсутствующие на стандартной клавиатуре (не говоря уже о том, что это может подтолкнуть седобородых хакеров рассказать вам одну забавную историю о языке APL). Поэтому лучше всего оставить символы Юникода исключительно для использования в комментариях и в литералах строк.

Литералы множеств

Множество элементов теперь можно определить, как литерал, заключив коллекцию значений в фигурные скобки { *элементы* }. Например:

```
days = { 'Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat', 'Sun' }
```

Этот синтаксис повторяет синтаксис использования функции `set()`:

```
days = set(['Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat', 'Sun'])
```

Генераторы множеств и словарей

Конструкция вида { *expr* `for` *x* `in` *s* `if` *condition* } – это генератор множества. Она применяет выражение *expr* ко всем элементам множества *s* и может использоваться на манер генераторов списков. Например:

```
>>> values = { 1, 2, 3, 4 }
>>> squares = {x*x for x in values}
>>> squares
{16, 1, 4, 9}
>>>
```

Конструкция вида { *kexpr*:*vexpr* `for` *k*,*v* `in` *s* `if` *condition* } – это генератор словаря. Она выполняет указанную операцию над всеми ключами и значениями в последовательности *s* кортежей (*key*, *value*) и возвращает словарь. Ключами нового словаря будут результаты выражения *kexpr*, а значениями – результаты выражения *vexpr*. Данную конструкцию можно рассматривать, как улучшенную версию функции `dict()`.

Чтобы проиллюстрировать использование этой конструкции, предположим, что у нас имеется файл `'prices.dat'` с котировками акций:

```
GOOG 509.71
YHOO 28.34
IBM 106.11
MSFT 30.47
AAPL 122.13
```

Ниже приводится программа, которая читает содержимое файла в словарь, отображающий имена компаний в стоимость акций, используя генератор словаря:

```
fields = (line.split() for line in open("prices.dat"))
prices = {sym:float(val) for sym,val in fields}
```

В следующем примере реализовано приведение всех символов ключей этого словаря к нижнему регистру:

```
d = {sym.lower():price for sym,price in prices.items()}
```

В следующем примере создается словарь, который содержит названия компаний, цена акций которых превышает $100.00:

```
d = {sym:price for sym,price in prices.items() if price >= 100.0}
```

Расширенная операция распаковывания итерируемых объектов

В Python 2 элементы в итерируемых объектах могут распаковываться в отдельные переменные, как показано ниже:

```
items = [1,2,3,4]
a,b,c,d = items   # Распакует элементы в переменные
```

Чтобы выполнить распаковывание, количество переменных должно точно соответствовать количеству распаковываемых элементов.

В Python 3, чтобы распаковать лишь некоторые элементы последовательности, можно использовать имя переменной с шаблонным символом, куда будут сохраняться оставшиеся значения в виде списка. Например:

```
a,*rest = items     # a = 1, rest = [2,3,4]
a,*rest,d = items   # a = 1, rest = [2,3], d = 4
*rest, d = items    # rest = [1,2,3], d = 4
```

В этих примерах переменная, имени которой предшествует символ *, принимает все оставшиеся значения и сохраняет их в виде списка. Если лишние элементы отсутствуют в объекте, список может остаться пустым. Один из примеров использования такой возможности – циклический обход списков кортежей (или последовательностей), когда кортежи могут иметь различные размеры. Например:

```
points = [ (1,2), (3,4,"red"), (4,5,"blue"), (6,7) ]
for x,y, *opt in points:
    if opt:
        # Обработка дополнительных полей
    инструкции
```

В этой конструкции может употребляться не более одной переменной со звездочкой.

Нелокальные переменные

Вложенные функции могут изменять значения переменных в объемлющих функциях, используя объявление `nonlocal`. Например:

```
def countdown(n):
    def decrement():
        nonlocal n
        n -= 1
    while n > 0:
        print("T-minus", n)
        decrement()
```

В Python 2 вложенные функции могут только прочитать значение переменной, объявленной в объемлющей функции, но не могут изменить его. Объявление `nonlocal` обеспечивает такую возможность.

Аннотации функций

Аргументы и возвращаемое значение функции могут быть аннотированы произвольными значениями. Например:

```
def foo(x:1,y:2) -> 3:
    pass
```

Функции имеют атрибут `__annotations__`, который является словарем, отображающим имена аргументов в аннотированные значения. Специальный ключ `'return'` отображается в аннотированное возвращаемое значение. Например:

```
>>> foo.__annotations__
{'y': 4, 'x': 3, 'return': 5}
>>>
```

Интерпретатор никак не использует эти аннотации. В действительности, они могут иметь любые значения. Однако, как предполагается, эта информация может быть полезна в будущем. Например, можно было бы написать такой код:

```
def foo(x:int, y:int) -> str:
    инструкции
```

Аннотация не обязана быть представлена единственным значением. В качестве аннотации может использоваться любое выражение, допустимое в языке Python. Для переменных, обозначающих коллекции позиционных и именованных аргументов, используется тот же синтаксис. Например:

```
def bar(x, *args:"additional", **kwargs:"options"):
    инструкции
```

Следует отметить еще раз, что интерпретатор Python никак не использует аннотации. Как предполагается, они будут использоваться сторонними библиотеками и фреймворками для различных нужд, связанных с мета-

программированием. В качестве примеров можно назвать инструменты статического анализа, документирования, тестирования, механизмы перегрузки функций, маршаллинга, вызова удаленных процедур, интегрированные системы разработки, средства проверки соблюдения соглашений и так далее. Ниже приводится пример функции-декоратора, который проверяет аргументы и возвращаемые значения функций:

```
def ensure(func):
    # Извлечь аннотированные данные
    return_check = func.__annotations__.get('return',None)
    arg_checks   = [(name,func.__annotations__.get(name))
                    for name in func.__code__.co_varnames]

    # Создать обертку, которая будет проверять значения аргументов и
    # возвращаемый результат с помощью функций, указанных в аннотациях
    def assert_call(*args,**kwargs):
        for (name,check),value in zip(arg_checks,args):
            if check: assert check(value), "%s %s" % (name, check.__doc__)
        for name,check in arg_checks[len(args):]:
            if check: assert check(kwargs[name]), "%s %s" % (name,
                                                             check.__doc__)
        result = func(*args,**kwargs)
        assert return_check(result), "return %s" % return_check.__doc__
        return result

    return assert_call
```

Ниже приводится пример использования этого декоратора:

```
def positive(x):
    "must be positive"
    return x > 0

def negative(x):
    "must be negative"
    return x < 0

@ensure
def foo(a:positive, b:negative) -> positive:
    return a - b
```

Ниже приводятся несколько примеров вызова этой функции:

```
>>> foo(3,-2)
5
>>> foo(-5,2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "meta.py", line 19, in call
    def assert_call(*args,**kwargs):
AssertionError: a must be positive
(Перевод:
Трассировочная информация (самый последний вызов - самый нижний):
  Файл "<stdin>", строка 1, в <module>
  Файл "meta.py", строка 19, в вызове
    def assert_call(*args,**kwargs):
```

```
AssertionError: a должно быть положительным
)
>>>
```

Только именованные аргументы

Функцию можно определить так, что она будет принимать некоторые аргументы только как именованные аргументы. Достигается это за счет добавления дополнительных аргументов после первого аргумента со звездочкой в имени. Например:

```
def foo(x, *args, strict=False):
    инструкции
```

В вызове такой функции аргумент `strict` может быть указан только как именованный. Например:

```
a = foo(1, strict=True)
```

Любые дополнительные позиционные аргументы будут помещаться в кортеж `args` и не будут использоваться для установки значения аргумента `strict`. Если нежелательно, чтобы функция принимала переменное число аргументов, но желательно, чтобы некоторые аргументы принимались только как именованные, этого можно добиться, добавив одиночный символ `*` в список аргументов. Например:

```
def foo(x, *, strict=False):
    инструкции
```

Ниже приводится пример использования:

```
foo(1,True)        # Ошибка. TypeError: foo() takes 1 positional argument
                   #         TypeError: foo() принимает 1 позиционный аргумент
foo(1,strict=True) # Ok.
```

Объекты класса Ellipsis как выражения

Объект класса `Ellipsis` object (`...`) теперь может использоваться в качестве выражения. Это позволяет сохранять такие объекты в контейнерах и в переменных. Например:

```
>>> x = ...        # Присвоит объект класса Ellipsis
>>> x
Ellipsis
>>> a = [1,2,...]
>>> a
[1, 2, Ellipsis]
>>> ... in a
True
>>> x is ...
True
>>>
```

Способ интерпретации многоточий остается за использующим их приложением. Эта особенность позволяет использовать многоточие (`...`) как часть син-

таксиса в библиотеках и фреймворках (например, как шаблонный символ, как признак продолжения и для обозначения других подобных понятий).

Цепочки исключений

Теперь имеется возможность объединять исключения в цепочки. По сути это обеспечивает возможность передать в текущем исключении информацию о предыдущем исключении. Для явного объединения исключений используется инструкция `raise` с квалификатором `from`. Например:

```
try:
    инструкции
except ValueError as e:
    raise SyntaxError("Couldn't parse configuration") from e
```

В случае возбуждения исключения `SyntaxError` будет выведено следующее сообщение об ошибке, содержащее информацию о двух исключениях:

```
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
ValueError: invalid literal for int() with base 10: 'nine'

The above exception was the direct cause of the following exception:

Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
SyntaxError: Couldn't parse configuration
(Перевод:
Трассировочная информация (самый последний вызов - самый нижний):
  Файл "<stdin>", строка 2, в <module>
ValueError: недопустимый литерал в вызове int() с основанием 10: 'nine'

Исключение, приведенное выше, стало причиной следующего исключения:

Трассировочная информация (самый последний вызов - самый нижний):
  Файл "<stdin>", строка 4, в <module>
SyntaxError: Ошибка разбора файла с настройками
)
```

Объекты исключений обладают атрибутом `__cause__`, в котором сохраняется предыдущее исключение. Квалификатор `from` в инструкции `raise` устанавливает значение этого атрибута.

Более сложный пример объединения исключений связан с возбуждением исключения в обработчике другого исключения. Например:

```
def error(msg):
    print(m)  # Обратите внимание: преднамеренная ошибка (имя m не определено)

try:
    инструкции
except ValueError as e:
    error("Couldn't parse configuration")
```

Если попытаться выполнить этот фрагмент в Python 2, будет выведено сообщение о единственном исключении `NameError`, возникшем в функции

error(). В Python 3 предыдущее обрабатываемое исключение будет объединено с результатом. Например, будет получено следующее сообщение:

```
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
ValueError: invalid literal for int() with base 10: 'nine'

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
  File "<stdin>", line 2, in error
NameError: global name 'm' is not defined
(Перевод:
Трассировочная информация (самый последний вызов - самый нижний):
  Файл "<stdin>", строка 2, в <module>
ValueError: недопустимый литерал в вызове int() с основанием 10: 'nine'

В процессе обработки исключения, приведенного выше, возникло другое исключение:

Трассировочная информация (самый последний вызов - самый нижний):
  Файл "<stdin>", строка 4, в <module>
  Файл "<stdin>", строка 2, в error
NameError: глобальное имя 'm' не определено
)
```

При неявном объединении ссылка на предыдущее исключение сохраняется в атрибуте __context__ экземпляра последнего исключения.

Улучшенная реализация функции super()

Функция super() производит поиск методов в базовых классах. В Python 3 она может вызываться без аргументов. Например:

```
class C(A,B):
    def bar(self):
        return super().bar() # Вызовет метод bar() базового класса
```

В Python 2 пришлось бы вызывать эту функцию как super(C,self).bar(). Старый синтаксис по-прежнему поддерживается, но он выглядит менее удобочитаемым.

Улучшенные метаклассы

В Python 2 имеется возможность определять метаклассы, которые изменяют поведение других классов. Важной особенностью реализации является то, что обработка метаклассом выполняется только *после того*, как все тело класса будет выполнено интерпретатором. То есть после того, как интерпретатор выполнит все тело класса и заполнит словарь. Как только словарь будет заполнен, он передается конструктору метакласса (после того, как тело класса уже будет выполнено).

В Python 3 метаклассы могут выполнять дополнительные операции перед тем, как интерпретатор приступит к выполнению тела класса. Это достигается за счет определения в метаклассе специального метода с именем __pre-

pare__(*cls*, *name*, *bases*, ***kwargs*). В качестве результата данный метод должен возвращать словарь. Этот словарь будет заполняться интерпретатором по мере выполнения определений в теле класса. Ниже приводится пример, иллюстрирующий простейшую обработку:

```
class MyMeta(type):
    @classmethod
    def __prepare__(cls,name,bases,**kwargs):
        print("подготавливается",name,bases,kwargs)
        return {}
    def __new__(cls,name,bases,classdict):
        print("создается",name,bases,classdict)
        return type.__new__(cls,name,bases,classdict)
```

В Python 3 для определения метаклассов можно использовать альтернативный синтаксис. Например, ниже приводится определение класса, использующего метакласс MyMeta:

```
class Foo(metaclass=MyMeta):
    print("Начало определения методов")
    def __init__(self):
        pass
    def bar(self):
        pass
    print("Конец определения методов")
```

Выполнив предыдущий программный код, вы увидите следующие результаты, иллюстрирующие порядок выполнения:

```
подготавливается Foo () {}
Начало определения методов
Конец определения методов
создается Foo () {'__module__': '__main__',
                  'bar': <function bar at 0x3845d0>,
                  '__init__': <function __init__ at 0x384588>}
```

Методу __prepare__() метакласса передаются дополнительные именованные аргументы, которые указываются в списке базовых классов инструкции class. Например, при использовании инструкции class Foo(metaclass= MyMeta,spam=42,blah="Hello") метод MyMeta.__prepare__() получит именованные аргументы spam и blah. Это соглашение может использоваться для передачи метаклассу произвольных параметров настройки.

Чтобы заставить новый метод __prepare__() метаклассов произвести какие-то полезные действия, его можно обязать возвращать объект словаря, обладающий дополнительными свойствами. Например, если необходимо реализовать дополнительную обработку, которая должна выполняться в процессе определения класса, следует объявить класс, производный от класса dict, и переопределить в нем метод __setitem__(), в котором реализовать обработку операций присваивания элементам словаря класса. Следующий пример иллюстрирует такую возможность, объявляя метакласс, который сообщает об ошибках, если какой-либо метод или атрибут класса объявляется несколько раз.

```
class MultipleDef(dict):
    def __init__(self):
        self.multiple= set()
    def __setitem__(self,name,value):
        if name in self:
            self.multiple.add(name)
        dict.__setitem__(self,name,value)

class MultiMeta(type):
    @classmethod
    def __prepare__(cls,name,bases,**kwargs):
        return MultipleDef()
    def __new__(cls,name,bases,classdict):
        for name in classdict.multiple:
            print(name,"повторное определение")
        if classdict.multiple:
            raise TypeError("Присутствует несколько определений ")
            return type.__new__(cls,name,bases,classdict)
```

Если применить этот метакласс к определению другого класса, он будет сообщать об ошибке при обнаружении повторного определения любого из методов. Например:

```
class Foo(metaclass=MultiMeta):
    def __init__(self):
        pass
    def __init__(self,x): # Ошибка. __init__ повторное определение.
        pass
```

Типичные ошибки

Если вы планируете переход с версии Python 2 на версию Python 3, имейте в виду, что **Python 3 – это не только новый синтаксис и возможности языка.** При переделке основных частей ядра и библиотеки в некоторых случаях вносились неожиданные, не ставящие своей целью обеспечение совместимости, изменения. Некоторые аспекты Python 3 могут выглядеть, как ошибки для программистов, использующих **Python 2. Некоторые конструкции, «простые» в использовании в версии Python 2, теперь считаются недопустимыми.**

В этом разделе отмечаются некоторые наиболее серьезные ошибки, которые часто допускаются программистами, привыкшими использовать Python 2.

Текст и байты

В Python 3 проводится очень строгое разграничение между текстовыми строками (символы) и двоичными данными (байты). Такие литералы, как `"hello"`, представляют текстовые данные и хранятся в виде строк Юникода, а литералы, такие как `b"hello"`, представляет строки байтов (в данном случае строку, содержащую символы ASCII).

В Python 3 не допускается смешивать типы `str` и `bytes`. Например, если попытаться выполнить конкатенацию текстовой строки и строки байтов, будет возбуждено исключение `TypeError`. Это важное отличие от Python 2, где строка байтов автоматически была бы преобразована в строку Юникода.

Чтобы преобразовать текстовую строку *s* в строку байтов, необходимо вызвать метод *s*.encode(*encoding*). Например, вызов *s*.encode('utf-8') преобразует *s* в строку байтов в кодировке UTF-8. Чтобы строку байтов *t* преобразовать обратно в текстовую строку, необходимо вызвать метод *t*.decode(*encoding*). Методы `encode()` и `decode()` можно рассматривать как своеобразные операции «приведения типов» между текстовыми строками и строками байтов.

Наличие четкого разделения между текстом и двоичными данными можно считать удачным решением – правила смешивания строковых типов в Python 2 по меньшей мере были неясными и трудными для понимания. Однако одно из последствий такого разграничения состоит в том, что строки байтов в Python 3 не обеспечивают такой же гибкости в представлении «текста». Несмотря на наличие стандартных строковых методов, таких как `split()` и `replace()`, многие другие особенности строк байтов отличаются от тех, что имеются в Python 2. Например, если вывести строку байтов с помощью функции `print()`, вывод будет произведен с помощью функции `repr()` в виде b'*содержимое*'. Точно так же ни одна из строковых операций форматирования (`%`, `.format()`) не будет работать со строками байтов. Например:

```
x = b'Hello World'
print(x)                    # Выведет b'Hello World'
print(b"You said '%s'" % x) # TypeError: % operator not supported
                            # TypeError: оператор % не поддерживается
```

Отсутствие некоторых возможностей, присущих текстовым строкам, является потенциальной ловушкой для системных программистов. Несмотря на повсеместное распространение Юникода, во многих случаях бывает необходимо работать с текстом, представленным однобайтовыми символами, такими как ASCII. Чтобы избежать лишних сложностей, связанных с Юникодом, может появиться желание использовать тип `bytes`. Однако на самом деле это только осложнит обработку такого текста. Ниже приводится пример, иллюстрирующий потенциальную проблему:

```
>>> # Создать сообщение ответа, используя строки (Юникод)
>>> status = 200
>>> msg = "OK"
>>> proto = "HTTP/1.0"
>>> response = "%s %d %s" % (proto, status, msg)
>>> print(response)
HTTP/1.0 200 OK

>>> # Создать сообщение ответа, используя строки байтов (ASCII)
>>> status = 200
>>> msg = b"OK"
>>> proto = b"HTTP/1.0"
>>> response = b"%s %d %s" % (proto, status, msg)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
```

```
TypeError: unsupported operand type(s) for %: 'bytes' and 'tuple'
(Перевод:
Трассировочная информация (самый последний вызов – самый нижний):
  Файл "<stdin>", строка 1, в <module>
TypeError: неподдерживаемые типы операндов для %: 'bytes' и 'tuple'
)

>>> response = proto + b" " + str(status) + b" " + msg
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can't concat bytes to str
(Перевод:
Трассировочная информация (самый последний вызов – самый нижний):
  Файл "<stdin>", строка 1, в <module>
TypeError: невозможно объединить данные типов bytes и str
)

>>> bytes(status)
b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00....'

>>> bytes(str(status))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: string argument without an encoding
(Перевод:
Трассировочная информация (самый последний вызов – самый нижний):
  Файл "<stdin>", строка 1, в <module>
TypeError: не указана кодировка для строкового аргумента
)

>>> bytes(str(status),'ascii')
b'200'

>>> response = proto + b" " + bytes(str(status),'ascii') + b" " + msg
>>> print(response)
b'HTTP/1.0 200 OK'

>>> print(response.decode('ascii'))
HTTP/1.0 200 OK
>>>
```

В примере можно видеть, насколько четко в Python 3 проведена грань между текстом и двоичными данными. Даже операции, которые на первый взгляд должны выполняться очень просто, такие как преобразование целых чисел в символы ASCII, оказываются намного сложнее при работе с байтами.

В итоге, если предполагается выполнять какие-либо операции по обработке или форматированию текста, практически всегда лучше использовать стандартные текстовые строки. Если по завершении обработки необходимо получить строку байтов, для преобразования строки Юникода в строку байтов можно воспользоваться методом *s*.encode('latin-1').

Различие между текстом и двоичными данными может быть менее четким при использовании различных библиотечных модулей. Некоторые модули

одинаково хорошо работают как с текстовыми строками, так и со строками байтов, тогда как другие вообще не могут работать с байтами. В некоторых случаях поведение модуля может зависеть от типа входных данных. Например, функция `os.listdir(`*`dirname`*`)` возвращает имена файлов, которые могут быть успешно декодированы как строки Юникода, только если в аргументе *`dirname`* передается текстовая строка. Если в аргументе *`dirname`* передать строку байтов, все возвращаемые имена файлов будут представлены строками байтов.

Новая система ввода-вывода

В Python 3 была реализована совершенно новая система ввода-вывода, описание которой приводится в главе 19 «Службы операционной системы», в разделе с описанием модуля `io`. Новая система ввода-вывода также отражает существенные различия между текстом и двоичными данными, представленными в виде строк.

Если предполагается выполнять какие-либо операции ввода-вывода с текстовыми данными, в Python 3 потребуется открывать файлы в «текстовом режиме» и указывать кодировку, если кодировка по умолчанию (обычно UTF-8) по каким-либо причинам не подходит. При выполнении операций ввода-вывода с двоичными данными файлы должны открываться в «двоичном режиме», и операции должны выполняться над строками байтов. Типичной ошибкой является попытка вывести данные в файл или в поток ввода-вывода, открытый не в том режиме. Например:

```
>>> f = open("foo.txt","wb")
>>> f.write("Hello World\n")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/tmp/lib/python3.0/io.py", line 1035, in write
    raise TypeError("can't write str to binary stream")
TypeError: can't write str to binary stream
(Перевод:
Трассировочная информация (самый последний вызов - самый нижний):
  Файл "<stdin>", строка 1, в <module>
  Файл "/tmp/lib/python3.0/io.py", строка 1035, в write
    raise TypeError("невозможно записать данные типа str в двоичный поток")
TypeError: невозможно записать данные типа str в двоичный поток
)
>>>
```

Как известно, сокеты, каналы и некоторые другие типы потоков ввода-вывода всегда действуют в двоичном режиме. Одна из потенциальных проблем, которые могут возникнуть при работе с сетями, состоит в том, что многие протоколы (такие как HTTP, SMTP, FTP и другие) подразумевают работу с запросами/ответами в виде текста. Учитывая, что сокеты действуют в двоичном режиме, происходит смешивание операций ввода-вывода двоичных данных с обработкой текстовой информации, что может приводить к проблемам, о которых рассказывалось в предыдущем разделе. Будьте предельно внимательны.

Функции print() и exec()

Инструкции print и exec в версии Python 2 теперь стали функциями. Ниже приводятся некоторые примеры использования функции print(), в сравнении с использованием инструкции print в Python 2:

```
print(x,y,z)          # То же, что и : print x, y, z
print(x,y,z,end=' ')  # То же, что и : print x, y, z,
print(a,file=f)       # То же, что и : print >>f, a
```

Тот факт, что инструкция print превратилась в функцию print(), означает, что в случае необходимости ее можно заменить альтернативной реализацией.

Инструкция exec также превратилась в функцию exec(), но ее поведение в Python 3 несколько изменилось по сравнению с Python 2. Рассмотрим в качестве примера следующий фрагмент:

```
def foo():
    exec("a = 42")
    print(a)
```

В Python 2 при вызове функции foo() будет выведено число '42'. В Python 3 будет возбуждено исключение NameError с сообщением о том, что имя переменной a не определено. Проблема состоит в том, что функция exec(), как функция, оперирует только словарями, возвращаемыми функциями globals() и locals(). Однако словарь, возвращаемый функцией locals(), в действительности является *копией* словаря с локальными переменными. Присваивание, выполненное функцией exec(), изменит *копию* локальной переменной, но не саму локальную переменную. Ниже приводится один из способов решения этой проблемы:

```
def foo():
    _locals = locals()
    exec("a = 42",globals(),_locals)
    a = _locals['a']    # Извлечет значение переменной из копии
    print(a)
```

И вообще, не следует ожидать, что Python 3 будет поддерживать тот же уровень «волшебства», который поддерживался функциями exec(), eval() и execfile() в Python 2. Более того, функция execfile() вообще была убрана из языка (ее можно имитировать, передав открытый объект файла функции exec()).

Использование итераторов и представлений

В Python 3 итераторы и генераторы используются намного шире, чем в Python 2. Встроенные функции, такие как zip(), map() и range(), раньше возвращавшие списки, теперь возвращают итераторы. Чтобы создать список из значения, возвращаемого такой функцией, можно использовать функцию list().

В Python 3 используется несколько иной подход к извлечению ключей и значений из словарей. В Python 2, чтобы получить список ключей, значе-

ний или пар ключ/значение, можно было использовать такие методы, как *d*.keys(), *d*.values() или *d*.items() соответственно. В Python 3 эти методы возвращают так называемые объекты *представлений*. Например:

```
>>> s = { 'GOOG': 490.10, 'AAPL': 123.45, 'IBM': 91.10 }
>>> k = s.keys()
>>> k
<dict_keys object at 0x33d950>
>>> v = s.values()
>>> v
<dict_values object at 0x33d960>
>>>
```

Эти объекты поддерживают итерации, поэтому для просмотра их содержимого можно использовать цикл for. Например:

```
>>> for x in k:
...     print(x)
...
GOOG
AAPL
IBM
>>>
```

Объекты представлений всегда можно преобразовать обратно в словари, из которых они были получены. Одна из важных особенностей этих объектов состоит в том, что изменения в словаре приводят к изменению элементов, возвращаемых представлением. Например:

```
>>> s['ACME'] = 5612.25
>>> for x in k:
...     print(x)
...
GOOG
AAPL
IBM
ACME
>>>
```

Чтобы создать список ключей или значений словаря, достаточно просто воспользоваться функцией list(), например list(*s*.keys()).

Целые числа и целочисленное деление

В Python 3 больше нет типа int, соответствующего 32-битовым целым числам, и отдельного типа, соответствующего длинным целым числам. Теперь тип int представляет целые числа произвольной точности (подробности внутренней реализации этого типа недоступны пользователю).

Кроме того, операция целочисленного деления теперь всегда возвращает число с плавающей точкой. Например, результатом деления 3/5 будет число 0.6, а не 0. Преобразование в число с плавающей точкой производится, даже если результат может быть представлен целым числом. Например, результатом деления 8/2 будет число 4.0, а не 4.

Сравнивание

В Python 3 реализована более строгая процедура сравнивания значений. В Python 2 допускается сравнивать объекты, даже если такая операция сравнивания не имеет смысла. Например:

```
>>> 3 < "Hello"
True
>>>
```

В Python 3 такая операция приведет к исключению `TypeError`. Например:

```
>>> 3 < "Hello"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unorderable types: int() < str()
(Перевод:
Трассировочная информация (самый последний вызов - самый нижний):
  Файл "<stdin>", строка 1, в <module>
TypeError: несравнимые типы: int() < str()
)
>>>
```

Это не очень существенное изменение, но оно означает, что в Python 3 необходимо быть более внимательными к типам сравниваемых значений. Например, если вы собираетесь использовать метод `sort()` списка, все элементы этого списка должны быть совместимы с оператором `<`, в противном случае будет возбуждено исключение. В Python 2 операция была бы выполнена без сообщения об ошибке, обычно с бессмысленным результатом.

Итераторы и генераторы

В Python 3 были внесены незначительные изменения в протокол итераторов. Ранее, чтобы получить итератор объекта, вызывался метод `__iter__()` и затем в каждой итерации вызывался метод `next()`. Теперь метод `next()` переименован в `__next__()`. Для большинства пользователей эти изменения незаметны, за исключением случаев, когда пишется программный код, который выполняет итерации вручную, или когда объявляется собственный объект итератора. Не забудьте изменить имя метода `next()` в своих классах. Для обеспечения переносимости используйте встроенную функцию `next()`, которая вызывает метод `next()` или `__next__()`.

Имена файлов, аргументов и переменных окружения

В Python 3 имена файлов, аргументов командной строки в переменной `sys.argv` и переменных окружения в `os.environ` могут интерпретироваться как строки Юникода, в зависимости от региональных настроек системы. Единственная проблема состоит в том, что использование Юникода в пределах операционной системы может поддерживаться недостаточно полно. Например, во многих системах технически может быть возможным указывать имена файлов, параметры командной строки и имена переменных окружения только в виде последовательностей байтов, которые не соответствуют допустимой кодировке Юникода. Такие ситуации на практике

встречаются достаточно редко, тем не менее об этом следует помнить при использовании языка Python для решения задач системного администрирования. Как уже отмечалось ранее, подобные проблемы легко можно решить, передавая имена файлов и каталогов в виде строк байтов. Например, `os.listdir(b'/foo')`.

Реорганизация библиотеки

В Python 3 были реорганизованы и переименованы некоторые части стандартной библиотеки. Наиболее заметные изменения произошли в коллекции модулей, связанных с сетевыми взаимодействиями и поддержкой форматов данных, используемых в Интернете. Кроме того, из библиотеки было исключено большое количество устаревших модулей (например, `gopherlib`, `rfc822` и другие).

Теперь стандартной практикой стало использовать в именах модулей только символы нижнего регистра. Некоторые модули, такие как `ConfigParser`, `Queue` и `SocketServer`, были переименованы в `configparser`, `queue` и `socketserver` соответственно. Вы также должны стараться следовать этому соглашению при создании своих модулей.

Для реорганизации программного кода, ранее содержавшегося в разрозненных модулях, были созданы новые пакеты. Например, был создан пакет `http`, включающий в себя все модули, используемые для разработки серверов HTTP; все модули, предназначенные для парсинга разметки HTML, были собраны в пакете `html`; модули, реализующие технологии XML-RPC, были помещены в пакет `xmlrpc`, и так далее.

Если говорить о нерекомендуемых модулях, то в этой книге были представлены только те модули, которые в настоящее время используются в версиях Python 2.6 и Python 3.0. Если в существующих программах, написанных для Python 2, вам встретятся модули, не описанные здесь, велика вероятность что эти модули были объявлены нерекомендуемыми в пользу чего-то более современного. Например, в Python 3 отсутствует модуль `popen2`, часто используемый в Python 2 для запуска дочерних процессов. Вместо него желательно использовать модуль `subprocess`.

Импортирование по абсолютному пути

Реорганизация библиотеки коснулась также всех инструкций `import`, присутствующих в подмодулях пакетов, – в них теперь используются абсолютные имена. Подробнее об этом рассказывается в главе 8 «Модули, пакеты и дистрибутивы», но, в качестве примера, представьте, что у вас имеется пакет со следующей организацией:

```
foo/
   __init__.py
   spam.py
   bar.py
```

Если в файле `spam.py` использовать инструкцию `import bar`, будет возбуждено исключение `ImportError`, несмотря на то, что файл `bar.py` находится в том же самом каталоге. Чтобы импортировать этот подмодуль, в файле `spam.py`

необходимо использовать инструкцию `import foo.bar` или выполнить импорт относительно пакета, например `from . import bar`.

В Python 2 инструкция `import` всегда сначала проверяет текущий каталог в поисках соответствия, и только потом переходит к проверке других каталогов, указанных в `sys.path`.

Перенос программного кода и утилита 2to3

Перенос существующего программного кода с версии Python 2 на версию Python 3 является достаточно непростой задачей. Чтобы сразу прояснить ситуацию, нужно заметить, что не существует никаких волшебных модулей, флагов, переменных окружения или других инструментов, которые позволили бы интерпретатору Python 3 выполнять любые программы, написанные для Python 2. Тем не менее существуют определенные шаги, представленные ниже, которые помогут выполнить перенос программного кода.

Перенос программного кода на версию Python 2.6

При переносе программного кода на **Python 3 сначала рекомендуется выполнить** перенос на версию Python 2.6. Дело в том, что версия Python 2.6 не только имеет обратную совместимость с версией Python 2.5, но она также поддерживает некоторые новые особенности, присутствующие в Python 3. В качестве примеров можно привести улучшенный механизм форматирования строк, новый синтаксис исключений, литералы строк байтов, библиотеку ввода-вывода и абстрактные базовые классы. Благодаря этому программа, написанная для **Python 2, сможет использовать полезные преимущества** Python 3, даже если она еще не готова полностью к переносу на версию Python 3.

Еще одна причина в пользу переноса на версию Python 2.6 состоит в том, что при запуске с ключом `-3` интерпретатор **Python 2.6 выводит предупреждения**, когда встречается с использованием нерекомендуемых возможностей. Например:

```
bash-3.2$ python -3
Python 2.6 (trunk:66714:66715M, Oct 1 2008, 18:36:04)
[GCC 4.0.1 (Apple Computer, Inc. build 5370)] on darwin
Type “help”, “copyright”, “credits” or “license” for more information.
>>> a = { }
>>> a.has_key(‘foo’)
__main__:1: DeprecationWarning: dict.has_key() not supported in 3.x; use the in
operator
(Перевод:
__main__:1: DeprecationWarning: dict.has_key() не поддерживается в версии 3.x;
используйте оператор in
)
False
>>
```

Опираясь на эти предупреждения, необходимо внести изменения в программу и добиться их отсутствия при работе программы под управлением Python 2.6, после чего можно приступать к переносу программы на версию Python 3.

Полный охват программы тестами

В стандартной библиотеке Python имеются удобные модули, обеспечивающие возможность тестирования, включая `doctest` и `unittest`. Убедитесь, что программа полностью охвачена тестами, прежде чем приступать к переносу на версию Python 3. Если к данному этапу в программе вообще не было никаких тестов, самое время приступить к их созданию. Вам потребуется обеспечить максимально полный охват программы тестами и убедиться, что при запуске под управлением Python 2.6 все тесты выполняются без вывода предупреждений.

Использование утилиты 2to3

В состав Python 3 входит утилита с именем `2to3`, которая может помочь в переносе программного кода с версии Python 2.6 на версию Python 3. Обычно эта утилита находится в каталоге `Tools/scripts`, внутри дистрибутива с исходными текстами, и в большинстве систем устанавливается в тот же каталог, что и выполняемый файл `python3.0`. Это утилита командной строки, которая обычно запускается в командной оболочке UNIX или Windows.

В качестве примера рассмотрим следующую программу, использующую ряд нерекомендуемых особенностей.

```
# example.py
import ConfigParser

for i in xrange(10):
    print i, 2*i

def spam(d):
    if not d.has_key("spam"):
        d["spam"] = load_spam()
    return d["spam"]
```

Чтобы обработать эту программу утилитой `2to3`, выполните команду `2to3 example.py`. Например:

```
% 2to3 example.py
RefactoringTool: Skipping implicit fixer: buffer
RefactoringTool: Skipping implicit fixer: idioms
RefactoringTool: Skipping implicit fixer: set_literal
RefactoringTool: Skipping implicit fixer: ws_comma
--- example.py (original)
+++ example.py (refactored)
@@ -1,10 +1,10 @@
 # example.py
-import ConfigParser
+import configparser
```

```
-for i in xrange(10):
-    print i, 2*i
+for i in range(10):
+    print(i, 2*i)

 def spam(d):
-    if not d.has_key("spam"):
+    if "spam" not in d:
         d["spam"] = load_spam()
     return d["spam"]
RefactoringTool: Files that need to be modified:
RefactoringTool: example.py
```

Как видно из листинга, утилита `2to3` смогла идентифицировать части программы, которые могут считаться проблематичными и, вероятно, должны быть изменены. Они показаны в формате вывода утилиты `diff`. В данном примере мы применили утилиту `2to3` к единственному файлу, однако если ей передать имя каталога, она рекурсивно обойдет все файлы с исходными текстами на языке Python, присутствующие в дереве каталогов, и создаст отчет для каждого из них.

По умолчанию утилита `2to3` *не исправляет* обнаруженные проблемы в исходных файлах, она просто сообщает о фрагментах программного кода, которые требуется изменить. Основная сложность, с которой приходится сталкиваться при работе с утилитой `2to3`, заключается в недостатке информации. Например, взгляните на функцию `spam()`. Эта функция вызывает метод `d.has_key()`. В словарях метод `has_key()` был удален в пользу оператора `in`. Утилита `2to3` сообщает о необходимости замены вызова этого метода, но без дополнительной информации непонятно, действительно ли функция `spam()` работает со словарем. Может оказаться, что `d` является экземпляром какого-либо другого класса (например, базы данных), реализующего метод `has_key()`, и тогда замена вызова метода оператором `in` может оказаться ошибкой. Еще одна проблема, которая может возникнуть при использовании утилиты `2to3`, – это обработка строк байтов и строк Юникода. Учитывая, что Python 2 автоматически преобразует строки байтов в строки Юникода, достаточно часто можно встретить программный код, в котором эти два строковых типа смешиваются по невнимательности программиста. К сожалению, утилита `2to3` неспособна обнаружить эту проблему. Это одна из причин, почему так важно иметь как можно более полный охват программы тестами. И конечно же, многое зависит от особенностей самого приложения.

При желании утилиту `2to3` можно заставить исправлять отдельные несовместимости. Для начала следует получить список «исправлений», введя команду `2to3 -l`. Например:

```
% 2to3 -l
Available transformations for the -f/--fix option:
(Доступные исправления для параметра -f/--fix)
apply
basestring
buffer
```

```
callable
...
...
xrange
xreadlines
zip
```

Используя имена из этого списка, можно посмотреть, какие фактические изменения внесет выбранное исправление, просто выполнив команду `2to3 -f` *`исправление имя_файла`*. Если необходимо применить сразу несколько исправлений, их можно указать в виде последовательности из нескольких параметров `-f`, по одному для каждого исправления. Если вам потребуется внести фактические исправления в исходный файл, добавьте ключ `-w`: `2to3 -f` *`исправление`* `-w имя_файла`. Например:

```
% 2to3 -f xrange -w example.py
--- example.py (original)
+++ example.py (refactored)
@@ -1,7 +1,7 @@
 # example.py
 import ConfigParser
-for i in xrange(10):
+for i in range(10):
     print i, 2*i

 def spam(d):
RefactoringTool: Files that were modified:
RefactoringTool: example.py
```

Если заглянуть в файл `example.py` после выполнения этой операции, можно обнаружить, что вызов функции `xrange()` был замещен вызовом функции `range()` и больше никаких других изменений внесено не было. Резервная копия оригинального файла `example.py` сохраняется под именем `example.py.bak`.

Противоположностью ключу `-f` является ключ `-x`. Если выполнить команду `2to3 -x` *`исправление имя_файла`*, она проверит необходимость выполнения всех исправлений, кроме тех, что указаны в параметре `-x`.

Несмотря на то что утилита `2to3` может вносить исправления непосредственно в исходные файлы, тем не менее на практике этого лучше избегать. Имейте в виду, что интерпретация программного кода не может выполняться абсолютно точно, и поэтому утилита `2to3` не всегда «права в своем выборе». При переносе программного кода всегда лучше подходить к этой проблеме методично и последовательно, а не полагаться на авось и на то, что все «заработает» чудесным образом.

Утилита `2to3` обладает парой дополнительных параметров, которые могут пригодиться. Ключ `-v` включает режим подробного вывода, когда выводится дополнительная информация, которая может пригодиться при отладке. Ключ `-p` сообщает утилите `2to3`, что вы уже используете инструкцию `print` как функцию и что ее исправлять нет необходимости (включается инструкцией `from __future__ import print_statement`).

Практическая стратегия переноса

Ниже представлено описание практической стратегии переноса программного кода с версии Python 2 на версию Python 3. Напомню еще раз, что всегда лучше подходить к этой проблеме методично и не стремиться решить ее одним махом.

1. Убедитесь, что программа имеет достаточное количество модульных тестов и что все тесты безошибочно выполняются под управлением Python 2.
2. Перенесите программу и комплект тестов на версию Python 2.6 и убедитесь, что все тесты выполняются безошибочно.
3. Добавьте параметр `-3` в вызов интерпретатора Python 2.6. Изучите все предупреждения и измените программу так, чтобы она выполнялась и проходила все тесты без вывода предупреждений. Если все было сделано правильно, есть вероятность, что программа по-прежнему сможет работать под управлением Python 2.5 и, может быть, под управлением еще более ранних версий. На данном этапе вы просто убрали из программы всякий «хлам».
4. Создайте резервную копию программы (это должно быть сделано без лишних напоминаний).
5. Перенесите комплект модульных тестов на Python 3 и убедитесь, что среда тестирования работоспособна сама по себе. Отдельные модульные тесты будут терпеть неудачу (потому что перенос программы еще не был выполнен). Однако корректно написанные тесты должны быть способны обнаруживать отказы, не завершаясь аварийно из-за внутренних проблем самой системы тестирования.
6. С помощью утилиты `2to3` перенесите саму программу на Python 3. Запустите тестирование получившегося программного кода и исправьте все обнаруженные проблемы. Существуют различные стратегии выполнения этого этапа. Если вы считаете себя любимцем фортуны, можете позволить утилите `2to3` исправить все, что она посчитает нужным, и посмотреть, что из этого получится. Если вы более осторожны, исправьте с помощью утилиты `2to3` наиболее очевидные проблемы (инструкции `print` и `except`, функцию `xrange()`, имена библиотечных модулей и так далее), а затем постепенно разберитесь с оставшимися проблемами.

По окончании этого процесса ваша программа должна проходить все тесты и действовать точно так же, как и прежде.

Теоретически возможно структурировать программный код так, что он будет работать как под управлением Python 2.6, так и под управлением Python 3, без какого-либо вмешательства со стороны пользователя. Однако для этого придется очень тщательно следовать современным соглашениям по оформлению программного кода на языке Python; по крайней мере, вам совершенно точно придется добиться того, чтобы в процессе работы под управлением Python 2.6 не выводилось никаких предупреждений. Если для автоматического переноса программного кода требуется специфическое использование утилиты `2to3` (например, запуск с определенным на-

бором исправлений), вероятно, лучше будет написать сценарий на языке командной оболочки, который автоматически будет выполнять необходимые операции и избавит пользователей от необходимости самостоятельно пользоваться утилитой `2to3`.

Одновременная поддержка Python 2 и Python 3

Последний вопрос, касающийся переноса программного кода на Python 3, – возможно ли иметь *единый* программный код, который без дополнительных модификаций будет выполняться под управлением Python 2 и Python 3? В некоторых случаях такое вполне возможно, однако при этом возникает опасность, что программный код превратится в жуткую мешанину. Например, вам придется ликвидировать все инструкции `print` и убедиться, что ни одна из инструкций `except` не извлекает значения исключений (а вся необходимая информация извлекается с помощью функции `sys.exc_info()`). Некоторые особенности языка Python вообще невозможно заставить одновременно работать под управлением разных версий интерпретатора. Например, из-за различий в синтаксисе нет никакой возможности реализовать такой способ использования метаклассов, который был бы совместим с версиями Python 2 и Python 3.

Таким образом, если в вашем распоряжении имеется программный код, который должен работать под управлением **Python 2 и Python 3, в первую очередь** из него необходимо устранить использование любых нерекомендуемых возможностей, убедиться, что он работает под управлением Python 2.6, обеспечить максимально полный охват модульными тестами и попытаться подобрать такое подмножество параметров утилиты `2to3`, чтобы обеспечить возможность автоматического преобразования.

Примером варианта создания программной базы, единой для разных версий Python, может послужить комплект модульных тестов. Очень полезно иметь набор тестов, которые без дополнительных модификаций действуют в **Python 2.6 и Python 3, – для проверки корректности поведения приложения** после его преобразования с помощью утилиты `2to3`.

Участие в проекте

Как проект с открытым исходным кодом, Python продолжает развиваться благодаря содействию его пользователей. Для дальнейшего развития **Python 3 разработчикам особенно важно получать отчеты об ошибках, о проблемах** с производительностью и о других недостатках. Отправить отчет об ошибке можно на странице *http://bugs.python.org*. Не стесняйтесь – ваши отзывы помогут сделать Python еще лучше.

Алфавитный указатель

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

В

Г

Д

Ж

З

И

K

Л

М

H

П

P

C

T

Ш

Э

Ю