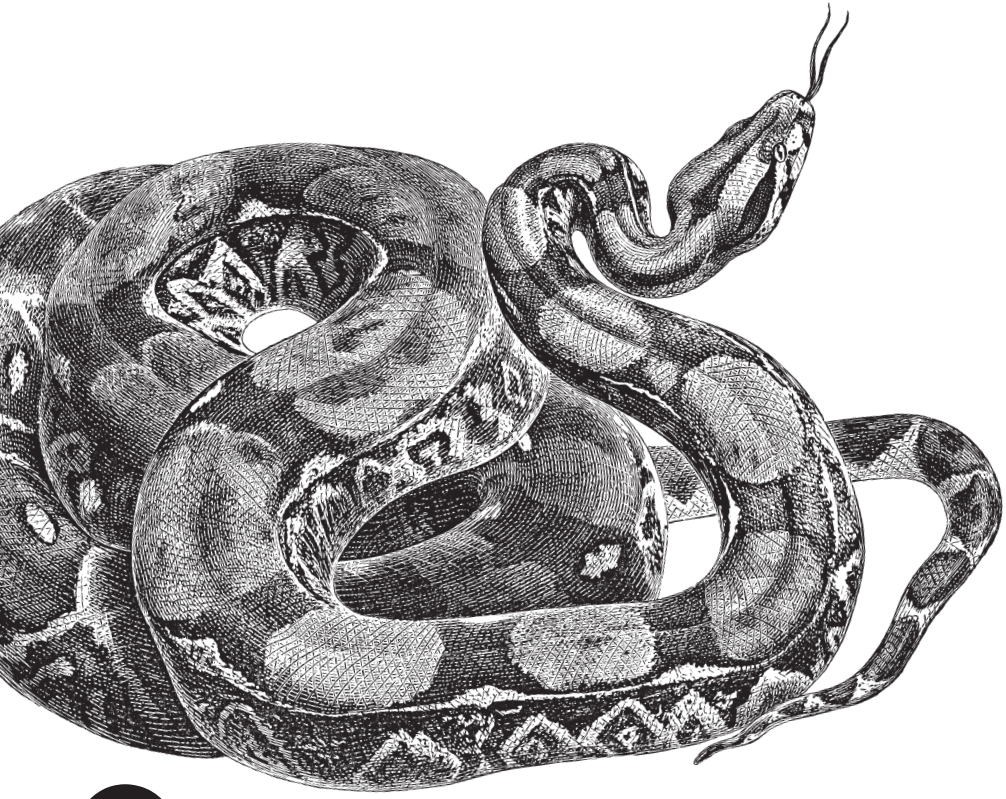
Эффективное решение проблем с помощью языка Python

Python

в системном администрировании UNIX и Linux



СИМВОЛ®

O'REILLY®

Ноа Гифт, Джереми М. Джонс

Python

for Unix and Linux System Administration

Noah Gift, Jeremy M. Jones

O'REILLY®

Python

в системном администрировании UNIX и Linux

Ноа Гифт и Джереми М. Джонс

Санкт-Петербург — Москва
2009

Ноа Гифт и Джереми М. Джонс

Python в системном администрировании UNIX и Linux

Перевод А. Киселева

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

Гифт Н., Джонс Д.
Python в системном администрировании UNIX и Linux – Пер. с англ. – СПб.: Символ-Плюс, 2009. – 512 с., ил.
ISBN 978-5-93286-149-3

Книга «Python в системном администрировании UNIX и Linux» демонстрирует, как эффективно решать разнообразные задачи управления серверами UNIX и Linux с помощью языка программирования Python. Каждая глава посвящена определенной задаче, например многозадачности, резервному копированию данных или созданию собственных инструментов командной строки, и предлагает практические методы ее решения на языке Python. Среди рассматриваемых тем: организация ветвления процессов и передача информации между ними с использованием сетевых механизмов, создание интерактивных утилит с графическим интерфейсом, организация взаимодействия с базами данных и создание приложений для Google App Engine. Кроме того, авторы книги создали доступную для загрузки и свободно распространяемую виртуальную машину на базе Ubuntu, включающую исходные тексты примеров из книги и способную выполнять примеры, использующие SNMP, IPython, SQLAlchemy и многие другие утилиты.

Издание рассчитано на широкий круг специалистов – всех, кто только начинает осваивать язык Python, будь то опытные разработчики сценариев на языках командной оболочки или относительно мало знакомые с программированием вообще.

ISBN 978-5-93286-149-3
ISBN 978-0-596-51582-9 (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.
Налоговая льгота – общероссийский классификатор продукции
ОК 005-93, том 2; 953000 – книги и брошюры.

Подписано в печать 12.01.2009. Формат 70×100¹/₁₆. Печать офсетная.
Объем 32 печ. л. Тираж 1000 экз. Заказ №
Отпечатано с готовых диапозитивов в ГУП «Типография «Наука»
199034, Санкт-Петербург, 9 линия, 12.

Я посвящаю эту книгу доктору Джозефу Е. Богену (Joseph E. Bogen),
моей матушке и моей супруге Леа – трем людям,
которые любили меня и верили в меня больше всех.

Ноа

Я посвящаю эту книгу моей жене Дебре и моим детям,
Зейну и Юстусу. Вы вдохновляли меня, дарили мне
свои улыбки и проявляли величайшее терпение,
пока я работал над этой книгой. Она по праву может
считаться настолько же вашей, насколько и моей.
Джереми

Оглавление

Вступительное слово

Я была приятно взволнована предложением выполнить предварительный обзор книги, посвященной использованию языка Python для нужд системного администрирования. Я вспомнила свои ощущения, когда впервые познакомилась с языком Python после многих лет программирования на других языках; это было похоже на свежесть весеннего ветра и согревающее тепло солнца после долгой зимы. Программирование на этом языке оказалось настолько необычайно простым и увлекательным делом, что мне удавалось заканчивать программы намного раньше, чем прежде.

Будучи системным администратором, я использовала язык Python в основном для решения задач системного и сетевого администрирования. Я заранее знала, насколько востребованной будет хорошая книга, посвященная применению языка Python в системном администрировании, и рада сказать, что это в полной мере относится к данной книге. Авторам, Ноа (Noah) и Джереми (Jeremy), удалось написать интересную и умную книгу о языке Python, который прочно обосновался в сфере системного администрирования. Я нахожу эту книгу полезной и увлекательной.

Две первые главы представляют собой введение в язык программирования Python для системных администраторов (и других), которые еще не знакомы с ним. Я отношу себя к программистам на языке Python среднего уровня, поэтому немало нового узнала из этой книги. Я полагаю, что даже искушенные программисты найдут здесь несколько новых приемов. Особенно я рекомендую прочитать главы, посвященные сетевому администрированию и управлению сетевыми службами, SNMP и управлению гетерогенными сетями, потому что в центре их внимания находятся нетривиальные и реальные задачи, с которыми системные администраторы сталкиваются ежедневно.

Элин Фриш (Æleen Frisch), июль 2008

Предисловие

Типографские соглашения

В этой книге используются следующие типографские соглашения:

Курсив

Курсивом выделяются новые термины, адреса URL, адреса электронной почты, имена файлов и их расширения.

`Моноширинный шрифт`

Используется для оформления листингов программ, для обозначения в тексте таких программных элементов, как имена переменных или функций, баз данных, типов данных, переменных окружения, инструкций, ключевых слов, утилит и модулей.

`Моноширинный жирный шрифт`

Используется для выделения команд и другого текста, который должен вводиться пользователем.

`Моноширинный курсив`

Используется для выделения текста, который пользователь должен заменить своими значениями или значениями, определяемыми контекстом.

Таким способом выделяются советы, предложения и примечания общего характера.

Таким способом выделяются предупреждения и предостережения.

Использование программного кода примеров

Данная книга призвана оказать вам помощь в решении ваших задач. Вообще вы можете свободно использовать примеры программного кода из этой книги в своих программах и в документации. Вам не нужно обращаться в издательство за разрешением, если вы не собираетесь воспроизводить существенные части программного кода. Например, если вы разрабатываете программу и используете в ней несколько отрывков программного кода из книги, вам не нужно обращаться за раз-

решением. Однако в случае продажи или распространения компакт-дисков с примерами из этой книги вам необходимо получить разрешение от издательства O'Reilly. Для цитирования данной книги или примеров из нее при разъяснении каких-либо вопросов получение разрешения не требуется. При включении существенных объемов программного кода примеров из этой книги в документацию на вашу продукцию вам *необходимо* получить разрешение издательства.

Мы приветствуем, но не требуем добавлять ссылку на первоисточник при цитировании. Под ссылкой на первоисточник мы подразумеваем указание авторов, издательства и ISBN. Например: «Python for Unix and Linux System Administration by Noah Gift and Jeremy M. Jones. Copyright 2008 Noah Gift and Jeremy M. Jones, 978-0-596-51582-9».

За получением разрешения на использование значительных объемов программного кода примеров из этой книги обращайтесь по адресу: *permissions@oreilly.com*.

Safari® Books Online

Если на обложке технической книги есть пиктограмма «Safari® Books Online», это означает, что книга доступна в Сети через O'Reilly Network Safari Bookshelf.

Safari предлагает намного лучшее решение, чем электронные книги. Это виртуальная библиотека, позволяющая без труда находить тысячи лучших технических книг, вырезать и вставлять примеры кода, загружать главы и находить быстрые ответы, когда требуется наиболее верная и свежая информация. Она свободно доступна по адресу *http://safari.oreilly.com*.

Отзывы и предложения

С вопросами и предложениями, касающимися этой книги, обращайтесь в издательство:

O'Reilly Media
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (в Соединенных Штатах Америки или в Канаде)
707-829-0515 (международный)
707-829-0104 (факс)

Список опечаток, файлы с примерами и другую дополнительную информацию вы найдете на сайте книги:

http://www.oreilly.com/9780596515829

Свои пожелания и вопросы технического характера отправляйте по адресу:

bookquestions@oreilly.com

Дополнительную информацию о книгах, обсуждения, Центр ресурсов издательства O'Reilly вы найдете на сайте:

http://www.oreilly.com

Благодарности

От Ноа

Лист благодарностей этой книги я хочу начать с доктора Джозефа Е. Богена (Joseph E. Bogen), потому что он – человек, который оказал наибольшее влияние на меня в тот момент, когда это было больше всего необходимо. Я встретил доктора Богена, когда работал в фирме Caltech, и он открыл мне глаза на другой мир, давая советы по жизненным ситуациям, психологии, неврологии, математике, по исследованиям в области сознания и во многих других областях. Это был умнейший человек, которого я когда-либо встречал, и я искренне любил его. Когда-нибудь я напишу книгу об этом опыте. Я опечален тем, что он не сможет прочитать ее; его смерть стала для меня большой утратой.

Я хочу сказать спасибо моей жене Леа (Leah), самой лучшей из всех женщин, встречавшихся мне. Без твоей любви и поддержки мне не удалось бы написать эту книгу. Ты терпелива, как ангел. Я надеюсь и дальше идти с тобой по жизни, я люблю тебя. Я также хочу поблагодарить моего сына Лиама (Liam), которому полтора года, за то, что терпел, пока я работал над этой книгой. Мне пришлось сильно урезать наши занятия музыкой и спортом, поэтому я должен вернуть тебе в два раза больше, мой маленький козленок.

Моей матушке: я люблю тебя и хочу сказать спасибо, что подбадривала меня все время.

Конечно же, я хочу сказать спасибо Джереми М. Джонсу (Jeremy M. Jones), моему соавтору – за то, что согласился написать эту книгу вместе со мной. Я думаю, из нас получилась отличная команда. У нас разные стили, но они прекрасно дополняют друг друга, и мы написали хорошую книгу. Вы дали мне много новых знаний о языке Python и были для меня хорошим партнером и другом. Спасибо!

Титус Браун (Titus Brown), которого теперь я должен бы называть доктором Брауном, был тем, кто разжег во мне первый интерес к языку Python, когда я встретил его в фирме Caltech. Это еще один пример того, какое важное значение может иметь один человек, и я рад считать его своим «старым» другом, которого не купишь ни за какие деньги. Он не уставал спрашивать меня: «Почему ты не используешь Python?». И однажды я решил попробовать. Если бы не Титус, я безусловно вернулся бы обратно к языкам Java и Perl. Вы можете почитать его блог по адресу: *http://ivory.idyll.org/blog*.

У Шеннона Беренса (Shannon Behrens) золотая голова, острый, как бритва, ум и потрясающее знание языка Python. Я познакомился

с Шенноном благодаря Титусу, и мы быстро подружились с ним. Шеннон – настоящий человек дела, во всех смыслах этого слова, и он дал мне огромный объем знаний о языке Python, можно даже сказать, гигантский. Его помощь во всем, что касалось языка Python, и в редактировании этой книги была просто неоценима, и я чрезвычайно обязан ему за это. Иногда я с ужасом думаю, какой могла бы быть эта книга без его помощи. Я не могу себе представить компанию, которая может упустить его, и я надеюсь помочь ему с его первой книгой. Наконец, он просто удивительный технический рецензент. Вы можете почитать его блог по адресу: *http://jjinux.blogspot.com/*.

Еще одним звездным техническим рецензентом был Дуг Хеллманн (Doug Hellmann). Сотрудничество с ним было исключительно плодотворным и полезным. Джереми и мне необычайно повезло в том, что нам удалось заполучить специалиста такого масштаба в качестве рецензента. Он не ограничился своим служебным долгом и стал настоящей движущей силой. Он был для нас неиссякаемым источником вдохновения, пока мы вместе с ним работали в компании Racemi. Вы можете почитать его блог по адресу: *http://blog.doughellmann.com/*.

Кому еще я хотел бы выразить свою признательность?

Скотту Лирсину (Scott Leerseen) – за обзор книги и за полезные советы в процессе работы над ней. Я получал огромное удовольствие от жарких споров, разгоравшихся вокруг фрагментов программного кода. Но помните – я всегда прав.

Альфредо Деза (Alfredo Deza) – за работу над настройкой виртуальной машины с Ubuntu, которая была необходима для работы над книгой. Твой опыт был для нас очень ценным.

Лайзе Дейли (Liza Daly) – за отзывы к первым черновым наброскам некоторых частей этой книги. Они были чрезвычайно полезными.

Джеффу Рашу (Jeff Rush) – за помощь и советы в работе с Buildout, Eggs и Virualenv.

Аарону Хиллегассу (Aaron Hillegass), владельцу замечательной обучающей компании Big Nerd Ranch, – за ценные советы и помощь во время работы. Мне крупно повезло, что посчастливилось встретиться с ним.

Марку Лутцу (Mark Lutz), под руководством которого я прошел курс обучения языку Python и который написал несколько замечательных книг по языку Python.

Членам сообщества Python в Атланте и участникам проекта PyAtl: *http://pyatl.org* – вы многому научили меня. Рик Коупленд (Rick Copeland), Рик Томас (Rick Thomas), Брендон Родс (Brandon Rhodes), Дерек Ричардсон (Derek Richardson), Джонатан Ла Кур (Jonathan La Cour), известный также под псевдонимом Mr. Metaclass, Дрю Смазерс (Drew Smathers), Кари Халл (Cary Hull), Бернард Меттьюс (Bernard Mat-

thews), Майкл Лангфорд (Michael Langford) и многие другие, кого я забыл упомянуть. Брендон и Рик Коупленд (Brandon and Rick Copeland) были в особенности полезны; они являются высококлассными программистами на языке Python. Вы можете почитать блог Брендона по адресу: *http://rhodesmill.org/brandon/*.

Григу Георгиу (Grig Gheorghiu) – за то, что делился с нами опытом системного администратора, за проверку советов и за то, что поддавал нам пинка, когда это было необходимо.

Моему работодателю, главному техническому директору и основателю компании Racemi, Чарльзу Уатту (Charles Watt). Я многому научился у вас и был рад, что вы знаете, когда какие кнопки нажимать. Помните, что я всегда готов написать для вас программу, пробежать 26-мильную дистанцию или проехать 200 миль на велосипеде – только сообщите мне, где и когда.

Доктору Нанде Ганесан (Nanda Ganesan), моему наставнику в аспирантуре Калифорнийского государственного университета в городе Лос-Анджелес (CSULA). Вы многому научили меня в области информационных технологий и в жизни и, кроме того, побуждали меня мыслить самостоятельно.

Доктору Синди Хейсс (Cindy Heiss) – моему профессору в мою бытность студентом. Вы приобщили меня к веб-разработке, научили верить в свои силы и, в конечном счете, оказали влияние на мою жизнь, спасибо!

Шелдону Блокбургеру (Sheldon Blockburger), позволившему мне попробовать свои силы в десятиборье в Калифорнийском государственном политехническом университете в городе Сан Луис Обиспо (Cal Poly SLO). Даже при том, что я не стал членом команды, вы развили во мне живой дух соперничества, качества борца и научили самодисциплине, предоставив мне самому отрабатывать забеги на короткие дистанции. И поныне еженедельные тренировки позволяют мне не потерять форму, в том числе и как программисту.

Брюсу Дж. Беллу (Bruce J. Bell), с которым я работал в Caltech. В течение нескольких лет совместной работы он учил меня программированию, делился своими знаниями операционной системы UNIX, и я очень признателен ему за это. С его статьями вы можете познакомиться по адресу: *http://www.ugcs.caltech.edu/~bruce/*.

Альберто Валезу (Alberto Valez), моему боссу в Sony Imageworks, – за то, что он был, пожалуй, лучшим боссом из всех, кто у меня когда-либо был, и за то, что предоставил мне возможность полностью автоматизировать мою работу.

Монтажеру фильмов Эду Фуллеру (Ed Fuller), который помогал мне советами и оставался отличным другом все это время.

Было много и других людей, оказывавших мне неоценимую помощь в работе, включая Дженнифер Девис (Jennifer Davis), еще одного друга по Caltech, которая предоставила несколько ценных отзывов; нескольких друзей и коллег по работе в компании Turner – Дуга Уэйка (Doug Wake), Уэйна Бланкарда (Wayne Blanchard), Сэма Олгуда (Sam Allgood), Дона Воравонга (Don Voravong); моих друзей и коллег по работе в Disney Feature animation, включая Шина Сомероффа (Sean Someroff), Грега Нигла (Greg Neagle) и Бобби Ли (Bobby Lea). Грег Нигл (Greg Neagle), в частности, очень многому меня научил в Mac OS X. Спасибо также Дж. Ф. Паниссету (J. F. Panisset), с которым я встретился в Sony Imageworks, учившему меня общим принципам разработки. И хотя теперь он главный технический директор, он мог бы считаться ценным кадром в любой компании.

Я хотел бы поблагодарить еще несколько человек, оказавших существенное содействие: Майка Вагнера (Mike Wagner), Криса МакДауэлла (Chris McDowell) и Шона Смута (Shaun Smoot).

Спасибо членам сообщества Python. В первую очередь спасибо Гвидо Ван Россуму (Guido van Rossum) за создание такого замечательного языка, за его качества настоящего лидера и за то, что был терпелив со мной, когда я обращался за советом по поводу этой книги. В сообществе Python есть большое число других знаменитостей, разрабатывающих инструменты, которыми я пользуюсь каждый день. Это Ян Бикинг (Ian Bicking), Фернандо Перез (Fernando Perez) и Вилле Вайнио (Ville Vainio), Майк Байер (Mike Bayer), Густаво Немейер (Gustavo Niemeyer) и другие. Спасибо Дэвиду Бизели (David Beazely) за его замечательную книгу и его фантастическое руководство «PyCon 2008 on Generators». Спасибо всем, кто пишет о языке Python и о системном администрировании. Ссылки на их работы вы сможете отыскать на странице *http://wiki.python.org/moin/systems_administration*. Спасибо также команде проекта Repoze: Тресу Сиверу (Tres Seaver) и Крису МакДонаху (Chris McDonough) (*http://repoze.org/index.html*).

Отдельная благодарность Филиппу Дж. Эби (Phillip J. Eby) за замечательный набор инструментальных средств, за проявленное терпение и советы по разделу, посвященному использованию библиотеки setuptools. Спасибо также Джиму Фултону (Jim Fulton) за то, что терпеливо отвечал на шквал моих вопросов по использованию ZODB и buildout. Особое спасибо Мартьяну Фассену (Martijn Fassen), который учил меня пользоваться такими продуктами, как ZODB и Grok. Если вам интересно заглянуть в будущее разработки веб-приложений на языке Python, обратите внимание на проект Grok: *http://grok.zope.org/*.

Спасибо сотрудникам журнала «Red Hat Magazine» – Джулии Брис (Julie Bryce), Джессике Гербер (Jessica Gerber), Баша Харрису (Bascha Harris) и Рут Сьюл (Ruth Suehle) за то, что позволили мне опробовать идеи, излагаемые в книге, в форме статей. Спасибо также Майку Мак-

Крери (Mike McCrery) из IBM Developerworks за то, что предоставил мне возможность опубликовать в форме статей некоторые идеи из книги.

Я хочу поблагодарить множество людей, которые в разные моменты моей жизни говорили, что мне что-то не по силам. Почти на каждом жизненном этапе я встречал людей, которые пытались отговорить меня: начиная с того, что я не смогу поступить в колледж, в который хотел бы поступить, и заканчивая тем, что я никогда не смогу изучать программирование. Спасибо вам за то, что давали мне дополнительный толчок к воплощению моих мечтаний. Люди способны выстроить свою жизнь, если они по-настоящему верят в себя; я мог бы посоветовать каждому пытаться сделать то, что он действительно хочет сделать.

Наконец, спасибо издательству O'Reilly и Татьяне Апанди (Tatiana Apandi) за то, что верили в мою способность написать книгу о применении языка Python в системном администрировании. Вы рискнули, поверили в меня и в Джереми, и я благодарю вас за это. Пусть ближе к концу книги Татьяна оставила издательство, чтобы воплотить свои мечты, тем не менее, мы продолжали чувствовать ее присутствие. Я также хотел бы отметить нового редактора Джулию Стил (Jilie Steele), которая была благожелательна и отзывчива. Вы привнесли целое море спокойствия, что лично я ценил очень высоко. В будущем я надеюсь еще услышать приятные новости от Джулии и буду счастлив снова работать с ней.

От Джереми

Длинный список благодарностей от Ноа заставил меня почувствовать себя неблагодарным человеком, потому что мой список не получится таким длинным, и обескураженным, так как он поблагодарил почти всех, кому мне тоже хотелось бы сказать спасибо.

В первую очередь я хотел бы вознести слова благодарности Господу Богу, с помощью которого я могу творить и без которого я ничего не смог бы сделать.

А в земном смысле в первую очередь я хотел бы поблагодарить мою супругу Дебру (Debra). Ты занимала детей другими делами, пока я работал над книгой. Ты сделала законом фразу: «Не беспокойте папу, он работает над своей книгой». Ты подбадривала меня, когда мне это было необходимо, и ты выделила мне пространство, которое мне так требовалось. Спасибо тебе. Я люблю тебя. Без тебя я не смог бы написать эту книгу.

Я также хотел поблагодарить моих славных детей, Зейна (Zane) и Юстуса (Justus) за их терпение по отношению к моей работе над книгой. Я пропустил большое число поездок с вами в парк Каменная Гора. Я по-прежнему укладывал вас спать, но я не оставался и не засыпал вместе с вами, как обычно. Последние несколько недель я пропускал шоу «Kid's Rock», которое выходит вечером по средам. Я пропустил так много, но вы терпеливо выдержали все это. Спасибо вам за ваше

терпение. И спасибо вам за то, что радовались, когда услышали, что я почти закончил книгу. Я люблю вас обоих.

Я хочу поблагодарить своих родителей, Чарльза и Линду Джонс (Charles and Linda Jones), за их поддержку моей работы над этой книгой. Но больше всего я хочу сказать им спасибо за то, что были для меня примером этики, за то, что научили меня работать над собой и с умом тратить деньги. Надеюсь, что все это я смогу передать своим детям, Зейну и Юстусу.

Спасибо Ноа Гифту (Noah Gift), моему соавтору, за то, что втянул меня в это дело. Оно оказалось тяжелым, тяжелее, чем я думал, и определенно одно из самых тяжелых, которое мне когда-либо приходилось делать. Когда вы работаете с человеком над чем-то, подобным книге, и под конец по-прежнему считаете его своим другом, я думаю, это достаточно характеризует его. Спасибо, Ноа. Эта книга не состоялась бы без тебя.

Я хочу поблагодарить нашу команду рецензентов. Ноа уже поблагодарил всех вас, но я хочу еще раз поблагодарить Дуга Хеллмана (Doug Hellman), Дженнифер Девис (Jennifer Davis), Шеннона Дж. Беренса (Shannon J. Behrens), Криса МакДауэлла (Chris McDowell), Титуса Брауна (Titus Brown) и Скотта Лирсина (Scott Leerseen). Вы удивительные люди. Бывали моменты, когда я заходил в тупик, и вы направляли мои мысли в нужное русло. Вы привнесли свое видение и помогли мне увидеть книгу с разных точек зрения. (В основном это относится к вам, Дженнифер. Если глава, посвященная обработке текста, принесет пользу системным администраторам, то только благодаря вам.) Спасибо вам всем.

Я хотел бы сказать спасибо нашим редакторам, Татьяне Апанди (Tatiana Apandi) и Джулии Стил (Julie Steele). Вы взяли на себя рутинный труд, освободив нас для работы над книгой. Вы обе облегчили нашу ношу.

Я также хочу выразить свою признательность Фернандо Перезу (Fernando Perez) и Вилле Вайнио (Ville Vainio) за потрясающие отзывы. Надеюсь, что мне удалось воздать должное IPython. И спасибо вам за IPython. Без него моя жизнь оказалась бы труднее.

Спасибо вам, Дункан МакГреггор (Duncan McGreggor), за помощь с примерами использования платформы Twisted. Ваши комментарии были чрезвычайно полезны. И спасибо, что вы продолжаете работать над этой замечательной платформой. Я надеюсь, что теперь буду использовать ее более широко.

Я благодарю Брема Мулинаара (Bram Moolenaar) и всех тех, кто когда-либо работал над редактором Vim. Почти все слова и теги XML, которые мне пришлось написать, были написаны с помощью Vim. В процессе работы над книгой я узнал несколько новых приемов и ввел их

в свой повседневный обиход. Редактор Vim позволил мне поднять мою производительность. Спасибо вам.

Я также хочу сказать спасибо Линусу Торвальдсу (Linus Torvalds), разработчикам Debian, разработчикам Ubuntu и всем тем, кто когда-либо работал над операционной системой Linux. Почти каждое слово, которое я напечатал, было напечатано в Linux. Вы обеспечили невероятную простоту настройки новых окружений и проверку различных идей. Спасибо вам.

Наконец, но ни в коем случае не меньше других, я хочу поблагодарить Гвидо ван Россума (Guido van Rossum) и всех тех, кто когда-либо работал над языком программирования Python. Я извлекал выгоду из вашего труда на протяжении нескольких последних лет. Два своих последних места работы я получил благодаря знанию языка Python. Язык Python и сообщество его поклонников не раз радовали меня с тех пор, как я начал использовать этот язык где-то в 2001–2002 годах. Спасибо вам. Python пришелся мне по душе.

1

Введение

Почему Python?

Если вы системный администратор, вам наверняка пришлось сталкиваться с Perl, Bash, ksh и некоторыми другими языками сценариев. Вы могли даже использовать один или несколько языков в своей работе. Языки сценариев часто позволяют выполнять рутинную, утомительную работу со скоростью и надежностью, недостижимой без них. Любой язык – это всего лишь инструмент, позволяющий выполнить работу. Ценность языка определяется лишь тем, насколько точно и быстро с его помощью можно выполнить свою работу. Мы считаем, что Python представляет собой ценный инструмент именно потому, что он дает возможность эффективно выполнять нашу работу.

Можно ли сказать, что Python лучше, чем Perl, Bash, Ruby или любой другой язык? На самом деле очень сложно дать такую качественную оценку, потому что всякий инструмент очень тесно связан с образом мышления программиста, использующего его. Программирование – это субъективный и очень личностный вид деятельности. Язык становится превосходным, только если он полностью соответствует потребностям программиста. Поэтому мы не будем доказывать, что язык Python лучше, но мы объясним причины, по которым мы считаем Python лучшим выбором. Мы также объясним, почему он так хорошо подходит для решения задач системного администрирования.

Первая причина, по которой мы считаем Python превосходным языком, состоит в том, что он очень прост в изучении. Если язык не способен быстро превратиться для вас в эффективный инструмент, его привлекательность резко падает. Неужели вы хотели бы потратить недели или месяцы на изучение языка, прежде чем вы окажетесь в состоянии написать на нем что-либо стоящее? Это особенно верно для системных администраторов. Если вы – системный администратор, проблемы могут

накапливаться быстрее, чем вы можете разрешать их. С помощью языка Python вы сумеете начать писать полезные сценарии буквально спустя несколько часов, а не дней или недель. Если язык не позволяет достаточно быстро приступить к написанию сценариев, это повод задуматься в целесообразности его изучения.

Однако язык, пусть и простой в изучении, но не позволяющий решать сложные задачи, также не стоит потраченных на него усилий. Поэтому вторая причина, по которой мы считаем Python превосходным языком программирования, заключается в том, что он позволяет решать такие сложные задачи, какие только можно вообразить. Вам требуется строку за строкой просматривать файлы журналов, чтобы выудить из них какую-то важную информацию? Язык Python в состоянии помочь решить эту задачу. Или вам требуется просмотреть файл журнала, извлечь из него определенные записи и сравнить обращения с каждого IP-адреса в этом файле с обращениями в каждом из файлов журналов (которые хранятся в реляционной базе данных) за последние три месяца, а затем сохранить результаты в реляционной базе данных? Вне всяких сомнений это можно реализовать на языке Python. Язык Python используется для решения весьма сложных задач, таких как анализ генных последовательностей, для обеспечения работоспособности многопоточных веб-серверов и сложнейших статистических вычислений. Возможно, вам никогда не придется решать подобные задачи, но будет совсем нелишним знать, что в случае необходимости язык поможет вам решать их.

Кроме того, если вы в состоянии выполнять сложнейшие операции, но удобство сопровождения программного кода оставляет желать лучшего, это плохой знак. Язык Python ликвидирует проблемы, связанные с сопровождением программного кода, и он действительно позволяет выражать сложные идеи простыми языковыми конструкциями. Простота программного кода – существенный фактор, который облегчает дальнейшее его сопровождение. Программный код на языке Python настолько прост, что позволяет возвращаться к нему спустя месяцы. И достаточно прост, чтобы можно было вносить изменения в программный код, который раньше нам не встречался. Таким образом, синтаксис и общие идиомы этого языка настолько ясные, краткие и простые, что позволяют работать с ним в течение длительных периодов времени.

Следующая причина, по которой мы считаем Python превосходным языком, заключается в высокой удобочитаемости программного кода. Блоки программного кода определяются по величине отступов. Отступы помогают взгляду следить за ходом выполнения программы. Кроме того, язык Python основан на «использовании слов». Под этим подразумевается, что хотя в языке Python используются свои специальные символы, основные его особенности в большинстве своем реализованы в виде ключевых слов или библиотек. Упор на слова, а не на специальные символы упрощает чтение и понимание программного кода.

Теперь, когда мы выявили некоторые преимущества языка Python, мы проведем сравнение нескольких фрагментов программного кода на языках Python, Perl и Bash. Попутно мы познакомимся еще с несколькими преимуществами языка Python. Ниже приводится простой пример на языке Bash, который выводит все возможные парные комбинации символов из набора `1`, `2` и символов из набора `a`, `b`:

```
#!/bin/bash

for a in 1 2; do
    for b in a b; do
        echo "$a $b"
    done
done
```

Вот эквивалентный фрагмент на языке Perl:

```
#!/usr/bin/perl

foreach $a ('1', '2') {
    foreach $b ('a', 'b') {
        print "$a $b\n";
    }
}
```

Это самый простой вложенный цикл. А теперь сравним эти реализации с циклом `for` в языке Python:

```
#!/usr/bin/env python

for a in [1, 2]:
    for b in ['a', 'b']:
        print a, b
```

Далее продемонстрируем использование условных инструкций в Bash, Perl и Python. Здесь используется простая условная инструкция `if/else`, с помощью которой выясняется – является ли заданный путь к файлу каталогом:

```
#!/bin/bash

if [ -d "/tmp" ] ; then
     echo "/tmp is a directory"
else
    echo "/tmp is not a directory"
fi
```

Ниже приводится эквивалентный сценарий на языке Perl:

```
#!/usr/bin/perl

if (-d "/tmp") {
    print "/tmp is a directory\n";
}
else {
```

```
    print "/tmp is not a directory\n";
}
```

А ниже – эквивалентный сценарий на языке Python:

```
#!/usr/bin/env python

import os
if os.path.isdir("/tmp"):
    print "/tmp is a directory"
else:
    print "/tmp is not a directory"
```

Еще один фактор, говорящий в пользу превосходства языка Python, – это простота поддержки объектно-ориентированного стиля программирования (ООП). А также то обстоятельство, что вас ничто не заставляет использовать ООП, если в этом нет необходимости. Но когда появляется потребность в нем, этот стиль оказывается чрезвычайно простым в применении. ООП позволяет легко и просто разделить проблему на составные функциональные части, объединенные в нечто под названием «объект». Язык Bash не поддерживает ООП, но Perl и Python поддерживают. Ниже приводится модуль на языке Perl с определением класса:

```
package Server;
use strict;

sub new {
    my $class = shift;
    my $self = {};
    $self->{IP} = shift;
    $self->{HOSTNAME} = shift;
    bless($self);
    return $self;
}

sub set_ip {
    my $self = shift;
    $self->{IP} = shift;
    return $self->{IP};
}

sub set_hostname {
    my $self = shift;
    $self->{HOSTNAME} = shift;
    return $self->{HOSTNAME};
}

sub ping {
    my $self = shift;
    my $external_ip = shift;
    my $self_ip = $self->{IP};
    my $self_host = $self->{HOSTNAME};
    print "Pinging $external_ip from $self_ip ($self_host)\n";
```

```
    return 0;
}

1;
```

И далее фрагмент, в котором он используется:

```
#!/usr/bin/perl

use Server;

$server = Server->new('192.168.1.15', 'grumbly');
$server->ping('192.168.1.20');
```

Программный код, в котором используется объектно-ориентированный модуль, достаточно прост. Однако на анализ самого модуля может потребоваться некоторое время, особенно если вы не знакомы с ООП или с особенностями реализации его поддержки в языке Perl.

Эквивалентный класс на языке Python и порядок его использования выглядят, как показано ниже:

```
#!/usr/bin/env python

class Server(object):
    def __init__(self, ip, hostname):
        self.ip = ip
        self.hostname = hostname
    def set_ip(self, ip):
        self.ip = ip
    def set_hostname(self, hostname):
        self.hostname = hostname
    def ping(self, ip_addr):
        print "Pinging %s from %s (%s)" % (ip_addr, self.ip, self.hostname)

if __name__ == '__main__':
    server = Server('192.168.1.20', 'bumbly')
    server.ping('192.168.1.15')
```

Примеры на языках Perl и Python демонстрируют некоторые из фундаментальных аспектов ООП, и вместе с тем они наглядно показывают различные особенности, которые используются в этих языках для достижения поставленной цели. Оба фрагмента решают одну и ту же задачу, но они отличаются друг от друга. Таким образом, если вы пожелаете использовать ООП, язык Python предоставит вам такую возможность. И вы достаточно легко и просто сможете включить его в свой арсенал.

Другое преимущество Python проистекает не из самого языка, а из его сообщества. В сообществе пользователей языка Python достигнуто единодушие по поводу способов решения определенных видов задач, которые вы должны (или не должны) использовать. Несмотря на то, что сам язык обеспечивает множество путей достижения одной и той же цели, соглашения, принятые в сообществе, могут рекомендовать

воздерживаться от использования некоторых из них. Например, инструкция `from module import *` в начале модуля считается вполне допустимой. Однако сообщество осуждает такое ее использование и рекомендует использовать либо инструкцию `import module`, либо инструкцию `from module import resource`. Импортирование всего содержимого модуля в пространство имен другого модуля может вызвать существенные осложнения, когда вы попытаетесь выяснить принцип действия модуля и узнать, где находятся вызываемые функции. Это конкретное соглашение поможет вам писать более понятный программный код, что позволит тем, кто будет сопровождать его, выполнять свою работу с большим удобством. Следование общепринятым соглашениям открывает вам путь к использованию наилучших приемов программирования. Мы считаем, что это идет только на пользу.

Стандартная библиотека языка Python – это еще одна замечательная особенность Python. Если применительно к языку Python вы услышите фразу «батарейки входят в комплект поставки», это лишь означает, что стандартная библиотека позволяет решать все виды задач без необходимости искать другие модули для этого. Например, несмотря на их отсутствие в самом языке, Python обеспечивает поддержку регулярных выражений, сокетов, нескольких потоков выполнения и функции для работы с датой/временем, синтаксического анализа документов XML, разбора конфигурационных файлов, функций для работы с файлами и каталогами, хранения данных, модульного тестирования, а также клиентские библиотеки для работы с протоколами http, ftp, imap, smtp и nntp и многое другое. Сразу после установки Python модули поддержки всех этих функциональных особенностей могут импортироваться вашими сценариями по мере необходимости. В вашем распоряжении имеются все перечисленные здесь функциональные возможности. Весьма впечатляет, что все это поставляется в составе Python и вам не требуется приобретать что-то еще. Все эти возможности будут чрезвычайно полезны вам при создании своих собственных программ на языке Python.

Простой доступ к огромному количеству пакетов сторонних производителей – еще одно важное преимущество Python. Помимо множества библиотек, входящих в стандартную библиотеку языка Python, существует большое число библиотек и утилит, доступных в Интернете, которые устанавливаются одной командой. В Интернете, по адресу *http://pypi.python.org*, существует каталог пакетов Python (Python Package Index, PyPI), где любой желающий может выкладывать свои пакеты в общее пользование. К моменту, когда писались эти строки, для загрузки было доступно более 3800 пакетов. В составе пакетов присутствуют IPython, который будет рассматриваться в следующей главе, Storm (модуль объектно-реляционного отображения, который будет рассматриваться в главе 12) и Twisted, сетевая платформа, которая будет рассматриваться в главе 5, – это только 3 названия из более чем

3800 пакетов. Начав пользоваться PyPI, вы обнаружите, что он совершенно необходим вам для поиска и установки полезных пакетов.

Многие из преимуществ языка Python проистекают из его базовой философии. Если в строке приглашения к вводу Python ввести команду `import this`**, перед вами появится так называемый «Дзен языка Python» Тима Петерса (Tim Peters):**

```
In [1]: import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
( Перевод:
Красивое лучше уродливого.
Явное лучше неявного.
Простое лучше сложного.
Сложное лучше усложненного.
Плоское лучше вложенного.
Разрежённое лучше плотного.
Удобочитаемость важна.
Частные случаи не настолько существенны, чтобы нарушать правила.
Однако практичность важнее чистоты.
Ошибки никогда не должны замалчиваться.
За исключением замалчивания, которое задано явно.
В случае неоднозначности сопротивляйтесь искушению угадать.
Должен существовать один, и, желательно, только один очевидный
способ сделать это.
Хотя он может быть с первого взгляда не очевиден, если ты не голландец.
Сейчас лучше, чем никогда.
Однако, никогда чаще лучше, чем *прямо* сейчас.
Если реализацию сложно объяснить, – это плохая идея.
Если реализацию легко объяснить, – это может быть хорошая идея.
Пространства имён – великолепная идея, их должно быть много!
)
```

Этот свод правил – не догма, следование которой считается обязательным на всех уровнях разработки языка, но сам дух его, кажется, пропитывает почти все, что происходит в языке и с языком. И мы считаем, что это замечательно. Возможно, именно поэтому мы день за днем стремимся использовать Python. Эта философия совпадает с тем, чего мы ждем от языка. И если она совпадает с вашими ожиданиями, значит язык Python будет хорошим выбором и для вас.

Мотивация

Если вы только что взяли эту книгу в свои руки в книжном магазине или читаете введение где-нибудь в Интернете, вы, возможно, задаете себе вопрос – насколько сложно будет изучить язык Python и стоит ли это делать. Несмотря на растущую популярность Python, многие системные администраторы до сих пор используют в своей работе только Bash и Perl. Если вы относитесь к их категории, вас наверняка обрадует тот факт, что язык Python очень прост в изучении. Хотя это вопрос личного мнения, но многие убеждены в том, что это самый простой язык для изучения и преподавания, и точка!

Если вы уже знакомы с языком Python или считаете себя гуру в программировании на другом языке, вы наверняка сможете прямо сейчас перейти к любой из следующих глав, не читая это введение, и сумеете разобраться в наших примерах. Мы старались создавать примеры, которые действительно помогут вам выполнять свою работу. В книге имеются примеры способов автоматического обнаружения и мониторинга подсетей с помощью SNMP, преобразования в интерактивную оболочку Python под названием IPython, организации конвейерной обработки данных, инструментов управления метаданными с помощью средств объектно-реляционного отображения, сетевых приложений, инструментов командной строки и многого другого.

Если у вас имеется опыт программирования на языке командной оболочки, вам тоже нет причин волноваться. Вы также легко и быстро освоите Python. Вам нужно лишь желание учиться, определенная доля любопытства и решимость – те же факторы, которые побудили вас взять в руки эту книгу и прочитать введение.

Мы понимаем, что среди вас есть и скептики. Возможно, часть из того, что вы слышали о программировании, напугала вас. Существует одно общее, глубоко неверное заблуждение, что программированию могут научиться не все, а только избранная таинственная элита. На самом деле любой желающий может научиться программировать. Второе, не менее ложное заблуждение состоит в том, что только получение профессионального образования в области информатики открывает человеку путь к званию программиста. Однако у некоторых самых талантливых программистов нет диплома об образовании в данной области. Среди компетентных программистов на языке Python имеются люди

с профессиональной подготовкой в области философии, журналистики, диетологии и английского языка. Наличие специального образования не является обязательным требованием для освоения Python, хотя оно и не повредит.

Еще одно забавное и такое же неверное представление заключается в том, что программированию можно учиться только в подростковом возрасте. Хотя это позволяет хорошо себя чувствовать людям, которым посчастливилось встретить в юности кого-то, кто вдохновил их заняться программированием, тем не менее, это еще один миф. Очень полезно начинать изучение программирования в юном возрасте, но возраст не является препятствием к освоению языка Python. Освоение Python – это не «игрушка для молодежи», как иногда говорят. Среди разработчиков существует бесчисленное множество людей, которые осваивали программирование, будучи в возрасте старше 20, 30, 40 лет и даже больше.

Если вы добрались до этого места, то надо заметить, что у вас, уважаемый читатель, есть одно преимущество, которое отсутствует у многих. Если вы решили взять в руки книгу, рассказывающую об использовании языка Python в системном администрировании UNIX и Linux, значит, вы уже представляете себе, как выполнять команды в командной оболочке. Это огромное преимущество для того, кто решил стать программистом на языке Python. Наличие знаний о способах выполнения команд в терминале – это все, что необходимо для этого введения в Python. Если вы уверены, что научитесь программированию на языке Python, тогда сразу же переходите к следующему разделу. Если у вас еще есть сомнения, тогда прочитайте этот раздел еще раз и убедите себя в том, что вы в силах овладеть программированием на языке Python. Это действительно просто, и если вы примете такое решение, оно изменит вашу жизнь.

Основы

Это введение в язык Python сильно отличается от любого другого, т. к. мы будем использовать интерактивную оболочку под названием IPython и обычную командную оболочку Bash. Вы должны будете открыть два окна терминалов, одно – с командной оболочкой Bash и другое – с интерактивной оболочкой IPython. В каждом примере мы будем сравнивать, как выполняются одни и те же действия с помощью Python и с помощью Bash. Для начала загрузите требуемую версию интерактивной оболочки IPython для своей платформы и установите ее. Получить ее можно на странице *http://ipython.scipy.org/moin/Download*. Если по каким-то причинам вы не можете получить и установить IPython, то можно использовать обычную интерактивную оболочку интерпретатора Python. Вы можете также загрузить копию виртуальной машины, включающую в себя все примеры программ из книги, а также предварительно настроенную и готовую к работе интер-

активную оболочку IPython. Вам достаточно просто ввести команду `ipython`, и вы попадете в строку приглашения к вводу.

Как только оболочка IPython будет установлена и запущена, вы должны увидеть примерно следующее:

```
[ngift@Macintosh-7][H:10679][J:0]# ipython
Python 2.5.1 (r251:54863, Jan 17 2008, 19:35:17)
Type "copyright", "credits" or "license" for more information.

IPython 0.8.2 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object'. ?object also works, ?? prints more.

In [1]:
```

Интерактивная оболочка IPython напоминает обычную командную оболочку Bash и может выполнять такие команды, как `ls`, `cd` и `pwd`, а в следующей главе приведены более подробные сведения о IPython. Эта глава посвящена изучению Python.

В окне терминала с интерактивной оболочкой Python введите следующую команду:

```
In [1]: print "I can program in Python"
I can program in Python
```

В окне терминала с командной оболочкой Bash введите следующую команду:

```
[ngift@Macintosh-7][H:10688][J:0]# echo "I can program in Bash"
I can program in Bash
```

В этих двух примерах не ощущается существенных различий между Python и Bash. Мы надеемся, что это поможет лишить Python части его загадочности.

Выполнение инструкций в языке Python

Если вам приходится тратить значительную часть времени на ввод команд в окне терминала, значит, вам уже приходилось выполнять инструкции и, возможно, перенаправлять вывод в файл или на вход другой команды UNIX. Рассмотрим порядок выполнения команды в Bash, а затем сравним его с порядком, принятым в Python. В окне терминала с командной оболочкой Bash введите следующую команду:

```
[ngift@Macintosh-7][H:10701][J:0]# ls -l /tmp/
total 0
-rw-r--r--  1 ngift  wheel  0 Apr  7 00:26 file.txt
```

В окне терминала с интерактивной оболочкой Python введите следующую команду:

```
In [2]: import subprocess

In [3]: subprocess.call(["ls","-l ","/tmp/"])
total 0
-rw-r--r--  1 ngift  wheel  0 Apr  7 00:26 file.txt
Out[3]: 0
```

Пример с командной оболочкой Bash не нуждается в пояснениях, так как это обычная команда `ls`, но, если ранее вам никогда не приходилось сталкиваться с программным кодом на языке Python, пример с интерактивной оболочкой Python наверняка покажется вам немного странным. Вы могли бы подумать: «Это еще что за команда `import subprocess`?». Одна из самых важных особенностей Python заключается в его возможности импортировать модули или другие файлы, содержащие программный код и используемые в новых программах. Если вам знакома инструкция `source` в Bash, то вы увидите определенное сходство с ней. В данном конкретном случае важно понять, что вы импортировли модуль subprocess и использовали его посредством синтаксической конструкции, показанной выше. Подробнее о том, как работают `subprocess` и `import`, мы расскажем позже, а пока не будем задумываться о том, почему эта инструкция работает, и обратимся к следующей строке:

```
subprocess.call(["команда", "аргумент", "другой_аргумент_или_путь"])
```

Из Python можно выполнить любую команду оболочки, как если бы она выполнялась командной оболочкой Bash. Учитывая эту информацию, можно сконструировать версию команды `ls` на языке Python. Откройте предпочтительный текстовый редактор или новую вкладку в окне терминала, создайте файл с именем *pyls.py* и сделайте его выполняемым с помощью команды `chmod +x pyls.py`. Содержимое файла приводится в примере 1.1.

Пример 1.1. Версия команды ls на языке Python

```
#!/usr/bin/env python
#Версия команды ls на языке Python

import subprocess

subprocess.call(["ls","-l"])
```

Если теперь запустить этот сценарий, вы получите тот же самый результат, что и при запуске команды `ls -l` из командной строки:

```
[ngift@Macintosh-7][H:10746][J:0]# ./pyls.py
total 8
-rwxr-xr-x  1 ngift  staff  115 Apr  7 12:57 pyls.py
```

Этот пример может показаться глупым (да он таким и является), но он наглядно демонстрирует типичное применение Python в системном администрировании. Язык Python часто используется для «обертывания» других сценариев или команд UNIX. Теперь вы уже смогли бы

начать писать некоторые несложные сценарии, помещая в файл команды одну за другой. Рассмотрим простые примеры, которые реализованы именно таким способом. Вы можете либо просто скопировать содержимое примера 1.2, либо выполнить сценарии *psysinfo.py* и *bashsysinfo.py*, которые можно найти в примерах к этой главе.

Пример 1.2. Сценарий получения информации о системе – Python

```
#!/usr/bin/env python
#Сценарий сбора информации о системе
import subprocess

#Команда 1
uname = "uname"
uname_arg = "-a"
print "Gathering system information with %s command:\n" % uname
subprocess.call([uname, uname_arg])

#Команда 2
diskspace = "df"
diskspace_arg = "-h"
print "Gathering diskspace information %s command:\n" % diskspace
subprocess.call([diskspace, diskspace_arg])
```

Пример 1.3. Сценарий получения информации о системе – Bash

```
#!/usr/bin/env bash
#Сценарий сбора информации о системе

#Команда 1
UNAME="uname -a"
printf "Gathering system information with the $UNAME command: \n\n"
$UNAME

#Команда 2
DISKSPACE="df -h"
printf "Gathering diskspace information with the $DISKSPACE command: \n\n"
$DISKSPACE
```

Если внимательно рассмотреть оба сценария, можно заметить, что они очень похожи. А если запустить их, будут получены идентичные результаты. Маленькое примечание: передавать в функции `subprocess.call` команду отдельно от аргумента совершенно необязательно. Можно использовать, например, такую форму записи:

```
subprocess.call("df -h", shell=True)
```

Все замечательно, но мы еще не объяснили, как действует инструкция `import` и что из себя представляет модуль `subprocess`. В версии сценария на языке Python мы выполнили импортирование модуля `subprocess`, т. к. он содержит программный код, позволяющий выполнять вызов команд системы.

Как уже упоминалось ранее, импортируя модуль `subprocess`, мы просто импортируем файл, содержащий необходимый нам программный код. Вы можете создать свой собственный модуль, или файл, и неоднократно использовать написанный вами программный код, импортируя его точно так же, как мы импортировали программный код из модуля `subprocess`. В импортировании нет ничего необычного, просто в результате этой операции вы получаете в свое распоряжение файл с некоторым программным кодом в нем. Одна из замечательных особенностей интерактивной оболочки IPython состоит в ее способности заглядывать внутрь модулей и файлов и получать списки доступных атрибутов. Если говорить терминами UNIX, это напоминает действие команды `ls` в каталоге */usr/bin*. Например, если вы оказались в новой системе, такой как Ubuntu или Solaris, а привыкли работать с Red Hat, то вы можете выполнить команду `ls` в каталоге */usr/bin*, чтобы узнать – имеется ли в наличии такой инструмент, как wget, curl или lynx. Если вы хотите воспользоваться инструментом, находящимся в каталоге */usr/bin*, можно просто ввести команду `/usr/bin/wget`, например.

Ситуация с модулями, такими как `subprocess`, очень похожа на описанную выше. В интерактивной оболочке IPython можно использовать функцию автодополнения, чтобы увидеть, какие инструменты доступны внутри модуля. Воспользуемся возможностью автодополнения и посмотрим, какие атрибуты имеются внутри модуля `subprocess`. Не забывайте, что модуль – это всего лишь файл с некоторым программным кодом внутри него. Ниже показано, что возвращает функция автодополнения в IPython для модуля `subprocess`:

```
In [12]: subprocess.
subprocess.CalledProcessError subprocess.__hash__      subprocess.call
subprocess.MAXFD              subprocess.__init__      subprocess.check_call
subprocess.PIPE               subprocess.__name__      subprocess.errno
subprocess.Popen              subprocess.__new__       subprocess.fcntl
subprocess.STDOUT             subprocess.__reduce__    subprocess.list2cmdline
subprocess.__all__            subprocess.__reduce_ex__ subprocess.mswindows
subprocess.__builtins__       subprocess.__repr__      subprocess.os
subprocess.__class__          subprocess.__setattr__   subprocess.pickle
subprocess.__delattr__        subprocess.__str__       subprocess.select
subprocess.__dict__           subprocess._active       subprocess.sys
subprocess.__doc__            subprocess._cleanup      subprocess.traceback
subprocess.__file__           subprocess._demo_posix   subprocess.types
subprocess.__getattribute__   subprocess._demo_windows
```

Чтобы воспроизвести этот эффект, вам нужно просто ввести команду:

```
import subprocess
```

затем ввести:

```
subprocess.
```

и нажать клавишу Tab, чтобы активизировать функцию автодополнения, которая выведет список доступных атрибутов. В третьей колонке нашего примера можно заметить `subprocess.call`. Теперь, чтобы получить дополнительную информацию об использовании `subprocess.call`, введите команду:

```
In [13]: subprocess.call?

Type:        function
Base Class:  <type 'function'>
String Form: <function call at 0x561370>
Namespace:   Interactive
File: /System/Library/Frameworks/Python.framework/Versions/2.5/lib/
python2.5/
      subprocess.py
Definition:   subprocess.call(*popenargs, **kwargs)
Docstring:
    Run command with arguments. Wait for command to complete, then
    return the returncode attribute.
    (Запускает команду с аргументами. Ожидает ее завершения и возвращает
    атрибут returncode)

    The arguments are the same as for the Popen constructor. Example:
    (Аргументы те же, что и в конструкторе Popen. Например:)
    retcode = call(["ls", "-l"])
```

Символ вопросительного знака в данном случае трактуется как обращение к странице справочного руководства. Когда требуется узнать, как работает некоторый инструмент в системе UNIX, достаточно ввести команду:

```
man имя_инструмента
```

То же и с атрибутом внутри модуля, таким как `subprocess.call`. Когда в оболочке IPython после имени атрибута вводится вопросительный знак, выводится документация, которая была включена в атрибут. Если подобную операцию выполнить с атрибутами из стандартной библиотеки, вы сможете обнаружить достаточно полезную информацию по их использованию. Имейте в виду, что существует также возможность обратиться к документации с описанием стандартной библиотеки языка Python.

Когда мы смотрим на это описание, в стандартный раздел «Docstring», мы видим пример использования атрибута `subprocess.call` и описание того, что он делает.

Итог

Теперь вы обладаете объемом знаний, достаточным, чтобы называть себя программистом на языке Python. Вы знаете, как написать простейший сценарий на языке Python, как перевести сценарий с языка Bash на язык Python, и наконец, вы знаете, как отыскать описание мо-

дулей и атрибутов. В следующем разделе вы узнаете, как организовать эти простые последовательности команд в функции.

Использование функций в языке Python

В предыдущем разделе мы узнали, как выполняются инструкции, что само по себе весьма полезно, т. к. это означает, что мы в состоянии автоматизировать выполнение некоторых операций, которые раньше выполнялись вручную. Следующим шагом к нашему программному коду автоматизации будет создание функций. Если вы еще не знакомы с функциями в языке Bash или в каком-либо другом языке программирования, то просто представляйте их себе как мини-сценарии. Функции позволяют создавать блоки инструкций, которые работают в группе. Это немного похоже на сценарий Bash с двумя командами, написанный нами ранее. Одно из отличий состоит в том, что вы можете включить в сценарий множество функций. В конечном счете можно весь программный код сценария расположить в функциях и затем запускать эти мини-программы в нужное время в своем сценарии.

Теперь настало время поговорить об отступах. В языке Python строки, принадлежащие одному и тому же блоку программного кода, должны иметь одинаковые отступы. В других языках, таких как Bash, когда определяется функция, ее тело заключается в фигурные скобки. В языке Python все строки в скобках должны иметь одинаковые отступы. Это может сбивать с толку тех, кто только начинает изучать язык, но через некоторое время это войдет в привычку и вы заметите, что выполнение этого требования повышает удобочитаемость программного кода. Если при работе с какими-либо примерами из книги у вас появляются ошибки, убедитесь для начала, что в исходных текстах правильно соблюдены отступы. Обычно один шаг отступа принимают равным четырем пробелам.

Рассмотрим, как работают функции в языке Python и Bash. Если у вас по-прежнему открыта интерактивная оболочка IPython, вы можете не создавать файл сценария на языке Python, хотя это и не возбраняется. Просто введите следующий текст в строке приглашения оболочки IPython:

```
In [1]: def pyfunc():
...:     print "Hello function"
...:
...:

In [2]: pyfunc
Out[2]: <function pyfunc at 0x2d5070>

In [3]: pyfunc()
Hello function

In [4]: for i in range(5):
```

```
...:     pyfunc()
...:
...:
Hello function
Hello function
Hello function
Hello function
Hello function
```

В этом примере инструкция `print` помещена в функцию. Теперь можно не только вызвать эту функцию позднее, но и вызвать ее столько раз, сколько потребуется. В строке `[4]` была использована идиома (прием) для выполнения функции пять раз. Если раньше вам такой прием не встречался, постарайтесь понять, что он вызывает функцию пять раз.

То же самое можно сделать непосредственно в командной оболочке Bash. Ниже демонстрируется один из способов:

```
bash-3.2$ function shfunc()
> {
>     printf "Hello function\n"
> }
bash-3.2$ for (( i=0 ; i < 5 ; i++))
> do
>     shfunc
> done
Hello function
Hello function
Hello function
Hello function
Hello function
```

В примере на языке Bash была создана простая функции `shfunc`, которая затем была вызвана пять раз, точно так же, как это было сделано ранее с функцией в примере на языке Python. Примечательно, что в примере на языке Bash потребовалось больше «багажа», чтобы реализовать то же самое, что и на языке Python. Обратите внимание на отличия цикла `for` в языке Bash от цикла `for` в языке Python. Если это ваша первая встреча с функциями в Bash или Python, вам следует поупражняться в создании каких-нибудь других функций в окне IPython, прежде чем двигаться дальше.

В функциях нет ничего таинственного, и попытка написать несколько функций в интерактивной оболочке поможет ликвидировать налет таинственности в случае, если это ваш первый опыт работы с функциями. Ниже приводится пара примеров простых функций:

```
In [1]: def print_many():
...:     print "Hello function"
...:     print "Hi again function"
...:     print "Sick of me yet"
...:
```

```
...:
In [2]: print_many()
Hello function
Hi again function
Sick of me yet

In [3]: def addition():
...:     sum = 1+1
...:     print "1 + 1 = %s" % sum
...:
...:

In [4]: addition()
1 + 1 = 2
```

Итак, у нас за плечами имеется несколько простейших примеров кроме тех, что вы попробовали выполнить сами, не правда ли? Теперь мы можем вернуться к сценарию, который собирает информацию о системе и реализовать его с применением функций, как показано в примере 1.4.

Пример 1.4. Преобразованный сценарий сбора информации о системе на языке Python: pysysinfo_func.py

```
#!/usr/bin/env python
#Сценарий сбора информации о системе
import subprocess

#Команда 1
def uname_func():

    uname = "uname"
    uname_arg = "-a"
    print "Gathering system information with %s command:\n" % uname
    subprocess.call([uname, uname_arg])

#Команда 2
def disk_func():

    diskspace = "df"
    diskspace_arg = "-h"
    print "Gathering diskspace information %s command:\n" % diskspace
    subprocess.call([diskspace, diskspace_arg])

#Главная функция, которая вызывает остальные функции
def main():
    uname_func()
    disk_func()

main()
```

Учитывая наши эксперименты с функциями, можно сказать, что преобразование предыдущей версии сценария вылилось в то, что мы просто поместили инструкции внутрь функций и затем организовали их вызов с помощью главной функции. Если вы не знакомы с подобным

стилем программирования, тогда, возможно, вы не знаете, что это достаточно распространенный прием, когда внутри сценария создается несколько функций, а затем они вызываются из одной главной функции. Одна из множества причин для такой организации состоит в том, что, когда вы решите использовать этот сценарий с другой программой, вы сможете выбирать, вызывать ли функции по отдельности или с помощью главной функции. Суть в том, что решение принимается после того, как модуль будет импортирован.

Когда нет никакого управления потоком выполнения или главной функции, весь программный код выполняется немедленно, во время импортирования модуля. Это может быть и неплохо для одноразовых сценариев, но если вы предполагаете создавать инструменты многократного пользования, тогда лучше будет использовать функции, которые заключают в себе определенные действия, и предусматривать создание главной функции, которая будет выполнять всю программу целиком.

Для сравнения также используем функции для предыдущего сценария на языке Bash, выполняющего сбор информации о системе, как показано в примере 1.5.

Пример 1.5. Преобразованный сценарий сбора информации о системе на языке Bash: bashsysinfo_func.sh

```
#!/usr/bin/env bash
#Сценарий сбора информации о системе

#Команда 1
function uname_func ()
{
    UNAME="uname -a"
    printf "Gathering system information with the $UNAME command: \n\n"
    $UNAME
}

#Команда 2
function disk_func ()
{
    DISKSPACE="df -h"
    printf "Gathering diskspace information with the $DISKSPACE command:
\n\n"
    $DISKSPACE
}

function main ()
{
    uname_func
    disk_func
}

main
```

Взглянув на наш пример на языке Bash, можно заметить немало схожего с аналогичным ему сценарием на языке Python. Здесь также созданы две функции, которые затем вызываются из главной функции. Если это ваш первый опыт работы с функциями, то мы могли бы порекомендовать вам закомментировать вызов главной функции в обоих сценариях, поставив в начале строки символ решетки (#), и попробовать запустить их еще раз. На этот раз в результате запуска сценариев вы не должны получить ровным счетом ничего, потому что программа хотя и выполняется, но она не вызывает две свои функции.

Теперь вы можете считать себя программистом, способным писать простые функции на обоих языках, Bash и Python. Программисты учатся работая, поэтому сейчас мы настоятельно рекомендуем вам изменить в обеих программах, на языке Bash и Python, вызовы системных команд своими собственными. Прибавьте себе несколько очков, если вы добавили в сценарии несколько новых функций и предусмотрели их вызов из главной функции.

Повторное использование программного кода с помощью инструкции import

Одна из проблем с освоением чего-либо нового состоит в том, что если это новое достаточно абстрактная вещь, бывает очень сложно найти ей применение. Когда в последний раз вам приходилось применять знание математики, полученное в средней школе, в продуктовом магазине? В предыдущих примерах было показано, как создавать функции, которые представляют альтернативу простому последовательному выполнению команд оболочки. Мы также сообщили, что модуль – это обычный сценарий или некоторое количество строк программного кода в файле. В этом подходе нет ничего сложного, но программный код должен быть организован определенным способом, чтобы его можно было повторно использовать в будущих программах. В этом разделе мы покажем вам, почему это так важно. Давайте импортируем оба предыдущих сценария сбора информации о системе и выполним их.

Откройте окна с IPython и Bash, если вы закрыли их, чтобы мы могли быстро продемонстрировать, почему функции играют такую важную роль с точки зрения повторного использования программного кода. Один из наших первых сценариев на языке Python представлял собой простую последовательность команд в файле с именем *pysysinfo.py*. В языке Python файл является модулем и наоборот, поэтому мы можем импортировать этот файл сценария в оболочку IPython. Обратите внимание, вы никогда не должны указывать расширение *.py* файла в инструкции импорта. Фактически попытка импорта окончится неудачей, если расширение будет указано. Итак, мы выполнили импорт сценария на ноутбуке Hoa Macbook Pro:

```
In [1]: import pysysinfo
Gathering system information with uname command:

Darwin Macintosh-8.local 9.2.2 Darwin Kernel Version 9.2.2: /
  Tue Mar 4 21:17:34 PST 2008; root:xnu-1228.4.31~1/RELEASE_I386 i386
Gathering diskspace information df command:

Filesystem     Size   Used  Avail Capacity Mounted on
/dev/disk0s2   93Gi   88Gi  4.2Gi    96%   /
devfs         110Ki  110Ki    0Bi   100%   /dev
fdesc         1.0Ki  1.0Ki    0Bi   100%   /dev
map -hosts      0Bi    0Bi    0Bi   100%   /net
map auto_home   0Bi    0Bi    0Bi   100%   /home
/dev/disk1s2  298Gi  105Gi  193Gi    36%   /Volumes/Backup
/dev/disk2s3  466Gi  240Gi  225Gi    52%   /Volumes/EditingDrive
```

Ух ты! Выглядит круто, правда? Когда импортируется файл, содержащий программный код на языке Python, он тут же выполняется. Но в действительности за всем этим кроется несколько проблем. Если вы планируете запускать такой программный код на языке Python, его всегда придется запускать из командной строки как часть сценария или программы, которую вы пишете. Операция импорта должна помочь в воплощении идеи «повторного использования программного кода». Но вот что интересно: как быть, если нам потребуется получить только информацию о распределении дискового пространства? В данном сценарии это невозможно. Именно для этого используются функции. Они позволяют контролировать, когда и какие части программы должны выполняться, чтобы она не выполнялась целиком, как в примере выше. Если импортировать сценарий, где эти команды оформлены в виде функций, можно увидеть, что мы имеем в виду.

Ниже приводится результат импортирования сценария в терминале IPython:

```
In [3]: import pysysinfo_func
Gathering system information with uname command:

Darwin Macintosh-8.local 9.2.2 Darwin Kernel Version 9.2.2: /
  Tue Mar 4 21:17:34 PST 2008; root:xnu-1228.4.31~1/RELEASE_I386 i386
Gathering diskspace information df command:

Filesystem     Size   Used  Avail Capacity Mounted on
/dev/disk0s2   93Gi   88Gi  4.2Gi    96%   /
devfs         110Ki  110Ki    0Bi   100%   /dev
fdesc         1.0Ki  1.0Ki    0Bi   100%   /dev
map -hosts      0Bi    0Bi    0Bi   100%   /net
map auto_home   0Bi    0Bi    0Bi   100%   /home
/dev/disk1s2  298Gi  105Gi  193Gi    36%   /Volumes/Backup
/dev/disk2s3  466Gi  240Gi  225Gi    52%   /Volumes/EditingDrive
```

Этот результат ничем не отличается от того, что был получен при использовании сценария без функций. Если вы озадачены, – это хороший знак. Чтобы понять, почему был получен тот же самый результат, дос-

таточно заглянуть в исходный программный код. Откройте сценарий *pysysinfo_func.py* в другой вкладке или в другом окне терминала и найдите строки:

```
#Главная функция, которая вызывает остальные функции
def main():
    uname_func()
    disk_func()

main()
```

Проблема в том, что функция `main`, созданная нами в конце предыдущего раздела, обернулась для нас некоторой неприятностью. С одной стороны, хотелось бы иметь возможность запускать сценарий из командной строки, чтобы получать полную информацию о системе, но с другой стороны, нам совсем не нужно, чтобы модуль выводил что-либо при импортировании. К счастью, потребность использовать модули как в виде сценариев, выполняемых из командной строки, так и в виде повторно используемых модулей достаточно часто встречается в языке Python. Решение этой проблемы состоит в том, чтобы определить, когда следует вызывать главную функцию, изменив последнюю часть сценария, как показано ниже:

```
#Главная функция, которая вызывает остальные функции
def main():
    uname_func()
    disk_func()

if __name__ == "__main__":
    main()
```

Эта «идиома» представляет прием, который обычно используется для решения данной проблемы. Любой программный код, входящий в состав блока этой условной инструкции, будет выполняться, только когда модуль запускается из командной строки. Чтобы убедиться в этом, измените окончание своего сценария или импортируйте исправленную его версию *pysysinfo_func_2.py*.

Если теперь вернуться к оболочке IPython и импортировать новый сценарий, вы должны увидеть следующее:

```
In [1]: import pysysinfo_func_2
```

На этот раз благодаря нашим исправлениям функция `main` вызвана не была. Итак, вернемся вновь к теме повторного использования программного кода: у нас имеется три функции, которые можно использовать в других программах или вызывать в интерактивной оболочке IPython. Вспомните: ранее мы говорили, что было бы неплохо иметь возможность вызвать только функцию, которая выводит информацию о распределении дискового пространства. Сначала необходимо вновь вернуться к одной из возможностей оболочки IPython, которую мы уже демонстрировали ранее. Вспомните, как мы использовали клавишу Tab

для получения полного списка атрибутов модуля, доступных для использования. Ниже показано, как выглядит этот список для нашего модуля:

```
In [2]: pysysinfo_func_2.
pysysinfo_func_2.__builtins__     pysysinfo_func_2.disk_func
pysysinfo_func_2.__class__        pysysinfo_func_2.main
pysysinfo_func_2.__delattr__      pysysinfo_func_2.py
pysysinfo_func_2.__dict__         pysysinfo_func_2.pyc
pysysinfo_func_2.__doc__          pysysinfo_func_2.subprocess
pysysinfo_func_2.__file__         pysysinfo_func_2.uname_func
pysysinfo_func_2.__getattribute__
pysysinfo_func_2.__hash__
```

В этом примере пока можно проигнорировать все имена, содержащие двойные символы подчеркивания, потому что они представляют специальные методы, описание которых выходит далеко за рамки этого введения. Поскольку IPython – это обычная командная оболочка, она обнаружила файл с расширением *.pyc*, содержащий скомпилированный байт-код Python. Отбросив все эти ненужные имена, можно заметить в списке имя `pysysinfo_func_2.disk_func`. Попробуем вызвать ее:

```
In [2]: pysysinfo_func_2.disk_func()
Gathering diskspace information df command:

Filesystem     Size   Used  Avail Capacity Mounted on
/dev/disk0s2   93Gi   88Gi  4.2Gi    96%   /
devfs         110Ki  110Ki    0Bi   100%   /dev
fdesc         1.0Ki  1.0Ki    0Bi   100%   /dev
map -hosts      0Bi    0Bi    0Bi   100%   /net
map auto_home   0Bi    0Bi    0Bi   100%   /home
/dev/disk1s2  298Gi  105Gi  193Gi    36%   /Volumes/Backup
/dev/disk2s3  466Gi  240Gi  225Gi    52%   /Volumes/EditingDrive
```

К настоящему моменту вы вероятно уже заметили, что функция «вызывается», или запускается, за счет указания круглых скобок «`()`» после ее имени. В данном случае мы использовали одну функцию из файла, содержащего три функции: `disk_func`, `uname_func` и, наконец, `main`. Ага! Мы все-таки сумели найти способ повторного использования нашего программного кода. Мы импортировали модуль, написанный нами ранее, и в интерактивной оболочке выполнили только ту его часть, которая была необходима. Безусловно, мы точно так же можем запустить и две другие функции. Давайте посмотрим на это:

```
In [3]: pysysinfo_func_2.uname_func()
Gathering system information with uname command:

Darwin Macintosh-8.local 9.2.2 Darwin Kernel Version 9.2.2:
  Tue Mar 4 21:17:34 PST 2008; root:xnu-1228.4.31~1/RELEASE_I386 i386

In [4]: pysysinfo_func_2.main()
Gathering system information with uname command:
```

```
Darwin Macintosh-8.local 9.2.2 Darwin Kernel Version 9.2.2:
  Tue Mar 4 21:17:34 PST 2008; root:xnu-1228.4.31~1/RELEASE_I386 i386
Gathering diskspace information df command:

Filesystem     Size   Used  Avail Capacity Mounted on
/dev/disk0s2   93Gi   88Gi  4.2Gi    96%   /
devfs         110Ki  110Ki    0Bi   100%   /dev
fdesc         1.0Ki  1.0Ki    0Bi   100%   /dev
map -hosts      0Bi    0Bi    0Bi   100%   /net
map auto_home   0Bi    0Bi    0Bi   100%   /home
/dev/disk1s2  298Gi  105Gi  193Gi    36%   /Volumes/Backup
/dev/disk2s3  466Gi  240Gi  225Gi    52%   /Volumes/EditingDrive
```

Если вы были внимательны, вы должны были заметить, что были вызваны обе оставшиеся функции. Не забывайте, что функция `main` запускает две другие функции.

Часто требуется взять из модуля только часть его программного кода и повторно использовать его в другом сценарии. Поэтому попробуем написать еще один сценарий, который использует только одну из функций. Пример такого сценария приводится в примере 1.6.

Пример 1.6. Повторное использование программного кода посредством импортирования: new_pysysinfo

```
#Очень короткий сценарий, использующий программный код из pysysinfo_func_2
from pysysinfo_func_2 import disk_func
import subprocess

def tmp_space():
    tmp_usage = "du"
    tmp_arg = "-h"
    path = "/tmp"
    print "Space used in /tmp directory"
    subprocess.call([tmp_usage, tmp_arg, path])

def main():
    disk_func()
    tmp_space()

if __name__ == "__main__":
    main()
```

В этом примере мы не только повторно использовали программный код, написанный ранее, но и использовали специальную синтаксическую конструкцию, которая позволяет импортировать только необходимые функции. Вся прелесть повторного использования программного кода состоит в том, что мы можем создавать совершенно новые программы, просто импортируя функции из других программ. Обратите внимание, что в функции `main` вызывается функция `disk_func()` из другого модуля, созданного нами, и новая, только что созданная, функция `tmp_space()` из этого файла.

В этом разделе мы узнали, насколько богатой может быть возможность повторного использования программного кода, и насколько просто ее использовать. В двух словах: вы помещаете одну-две функции в файл и затем, если вам требуется иметь возможность запускать сценарий из командной строки, используете специальную синтаксическую конструкцию `if__name__ == "__main__":`. После этого можно либо импортировать эти функции в оболочке IPython, либо просто использовать их в другом сценарии. Обладая этой информацией, вы становитесь по-настоящему опасными. Теперь вы можете создавать на языке Python довольно сложные модули и использовать их снова и снова при создании новых инструментов.

2

IPython

Одной из сильных сторон языка Python является его интерактивный интерпретатор, или оболочка. Оболочка обеспечивает возможность быстро проверить идею, протестировать функциональные возможности и интерфейсы модулей, с которыми вы работаете, и выполнить какие-либо однократные действия, для которых в другом случае пришлось бы писать сценарии из трех строк. Обычно при программировании на языке Python мы открываем текстовый редактор и интерактивную оболочку Python (в действительности оболочку IPython, но к этому мы вскоре еще вернемся), взаимодействуя с ними обоими, переключаясь взад-вперед между оболочкой и редактором, часто копируя фрагменты программного кода из одного окна в другое. При таком подходе мы можем быстро проверять работу программного кода в интерпретаторе и вставлять работоспособные и отлаженные фрагменты в текстовом редакторе.

В своей основе IPython представляет собой интерактивную оболочку Python. Эта удивительная оболочка обладает намного более широкими возможностями по сравнению со стандартной интерактивной оболочкой Python. Она позволяет создавать командные окружения, настраиваемые в весьма широких пределах; дает возможность встраивать интерактивную оболочку Python в любое приложение, написанное на языке Python и, с определенными ограничениями, может даже использоваться в качестве системной командной оболочки. В этой главе мы остановимся на использовании IPython с целью повышения эффективности решения задач, связанных с программированием на языке Python в *nix-оболочках.

За оболочкой IPython стоит весьма сплоченное сообщество. Вы можете подписаться на почтовую рассылку на странице *http://lists.ipython.scipy.org/mailman/listinfo/ipython-user*. Существует замечательная страница вики (wiki) *http://ipython.scipy.org/moin*. И как часть этой стра-

ницы – сборник рецептов *http://ipython.scipy.org/moin/Cookbook*. На любом из этих ресурсов вы можете читать информацию или выкладывать свою. Еще одна область, где вы можете попробовать приложить свои знания и умения, – это разработка IPython. Недавно разработка IPython была переведена на использование распределенной системы управления версиями исходных текстов, благодаря которой вы можете получить срез исходных текстов и приступить к их изучению. Если вы сделаете что-то, что может пригодиться другим, вы можете передать им свои изменения.

Установка IPython

Существует несколько вариантов установки IPython. Первый, самый традиционный, заключается в получении дистрибутива с исходными текстами. Страница, откуда можно загрузить IPython, находится по адресу *http://ipython.scipy.org/dist/*. К моменту написания этих строк последней была версия IPython 0.8.2 и близилась к завершению работа над версией 0.8.3. Чтобы установить IPython из исходных текстов, откройте страницу *http://ipython.scipy.org/dist/ipython-0.8.2.tar.gz* и загрузите файл *.tar.gz*. Распаковать этот файл можно с помощью команды `tar zxvf ipython-0.8.2.tar.gz`. В разархивированном каталоге будет

ПОРТРЕТ ЗНАМЕНИТОСТИ: ПРОЕКТ IPYTHON

Фернандо Перез (Fernando Perez)

Фернандо Перез – кандидат физико-математических наук, занимался разработкой числовых алгоритмов на кафедре прикладной математики университета в штате Колорадо. В настоящее время занимается научными изысканиями в институте неврологии имени Элен Уиллс (Helen Wills) в Калифорнийском университете в городе Беркли, сосредоточившись на разработке новых методов анализа для нужд моделирования мозговой деятельности и высокоуровневых инструментальных средств для научных вычислений. К моменту окончания обучения в аспирантуре он оказался вовлечен в разработку инструментальных средств для научных расчетов на языке Python. Он начал проект IPython в 2001 году, когда пытался разработать эффективный интерактивный инструмент для решения повседневных научных задач. Благодаря расширяющемуся кругу разработчиков этот проект продолжал расти и за эти годы превратился в инструмент, который будет полезен даже программистам, далеким от научных исследований.

ПОРТРЕТ ЗНАМЕНИТОСТИ: ПРОЕКТ IPYTHON

Вилле Вайнио (Ville Vainio)

Вилле Вайнио получил степень бакалавра информатики в 2003 году в Сатакунтском университете прикладных наук, на технологическом факультете в городе Пори, Финляндия. К моменту начала работы над этой книгой был нанят в качестве программиста в отдел смартфонов компании Digia Plc, где разрабатывает программное обеспечение на языке C++ для платформы Symbian OS, разработанной компаниями Nokia и UIQ. Во время учебы работал программистом в Cimcorp Oy, разрабатывал программное обеспечение на языке Python для взаимодействия с промышленными роботами. Вилле – давний приверженец IPython, а с 2006 года стал хранителем стабильной ветки IPython (серия 0.x). Его работа в проекте IPython началась с серии исправлений и улучшений IPython в части системной командной оболочки для Windows, и эксплуатация этой системной командной оболочки до сих пор остается в центре его внимания. Живет вместе со своей невестой в городе Пори, Финляндия, и пишет дипломный проект на получение степени магистра в местном филиале технологического университета г. Тампере, посвященный ILeo, который является программным мостом между IPython и Leo и превращает Leo в ноутбук с полноценной поддержкой IPython.

содержаться файл *setup.py*. Вызовите интерпретатор Python, передав ему имя файла *setup.py* с параметром `install` (например, `python setup.py install`). Эта команда установит библиотеки IPython в каталог *site-packages* и создаст сценарий `ipython` в каталоге *scripts*. В UNIX это обычно тот же каталог, где находится исполняемый файл интерпретатора `python`. Если вы используете `python`, установленный менеджером пакетов вашей системы, то этот файл (а, следовательно, и `ipython`) скорее всего будет находиться в каталоге */usr/bin*. Мы у себя установили IPython последней версии, которая еще находилась в разработке, поэтому в некоторых примерах вы будете видеть ее номер 0.8.3.

Второй вариант установки IPython заключается в установке пакета с помощью вашей системы управления пакетами. Для Debian и Ubuntu имеются доступные для установки пакеты *.deb*. Установка выполняется простой командой `apt-get install ipython`. В Ubuntu библиотеки IPython устанавливаются в соответствующее местоположение (*/usr/share/python-support/ipython*, куда помещается набор файлов *.pth* и символических ссылок, обеспечивающих корректную работу пакета). Кроме

того в каталог */usr/bin/python* устанавливается двоичный выполняемый файл `ipython`.

Третий вариант установки IPython заключается в использовании пакета Python. Вы могли даже не предполагать, что в Python существует такая вещь, как пакет, и, тем не менее, это так. Пакеты в языке Python – это файлы с расширением *.egg*, представляющие собой архивы формата ZIP. Пакеты можно устанавливать с помощью утилиты `easy_install`. Одна из замечательных особенностей утилиты `easy_install` заключается в том, что она проверяет центральный репозитарий пакетов и отыскивает необходимый пакет. За кулисами происходит несколько больше, чем только поиск пакета, однако для пользователя установка выполняется очень просто. Репозитарий называется каталогом пакетов Python (Python Package Index), или PyPI для краткости (хотя некоторые нежно называют его Python CheeseShop (сырная лавка Python)). Чтобы воспользоваться утилитой `easy_install`, необходимо зарегистрироваться в системе под учетной записью, которая обладает правом записи в каталог *site-packages*, и запустить команду `easy_install ipython`.

Четвертый вариант заключается в использовании IPython вообще без установки. «Что?», – можете вы спросить. Дело в том, что если загрузить дистрибутив с исходными текстами и просто запустить сценарий `ipython.py` из корневого каталога с набором файлов, то вы получите работающий экземпляр загруженной версии IPython. Этот способ подойдет тем, кто не желает загромождать свой каталог *site-packages*, хотя вам придется учитывать некоторые ограничения этого варианта. Если вы запускаете IPython из каталога, куда вы распаковали файл дистрибутива, и при этом не изменили переменную окружения `PYTHONPATH`, вы не сможете использовать этот продукт как библиотеку.

Базовые понятия

После того как оболочка IPython будет установлена, и вы в первый раз запустите команду `ipython`, вы увидите примерно следующее:

```
jmjones@dink:~$ ipython
**********************************************************************
Welcome to IPython. I will try to create a personal configuration directory
where you can customize many aspects of IPython's functionality in:
(Добро пожаловать в IPython. Я попробую создать персональный каталог
с настройками, где вы сможете настраивать разные аспекты IPython:)

/home/jmjones/.ipython

Successful installation!
(Установка выполнена благополучно!)

Please read the sections 'Initial Configuration' and 'Quick Tips' in the
IPython manual (there are both HTML and PDF versions supplied with the
```

```
distribution) to make sure that your system environment is properly
configured to take advantage of IPython's features.
(Пожалуйста, прочитайте разделы 'Initial Configuration' и 'Quick Tips'
в руководстве к IPython (в состав дистрибутива входит как HTML, так и PDF
версия), чтобы суметь убедиться, что системное окружение настроено должным
образом для использования IPython)

Important note: the configuration system has changed! The old system is
still in place, but its setting may be partly overridden by the settings in
"~/.ipython/ipy_user_conf.py" config file. Please take a look at the file
if some of the new settings bother you.
(Важное примечание: файл с настройками системы был изменен! Прежняя система
осталась на месте, но ее настройки могут оказаться частично переопределенными
настройками в файле "~/.ipython/ipy_user_conf.py". Пожалуйста, загляните
в этот файл, если новые значения параметров представляют для вас интерес.)

Please press <RETURN> to start IPython.
(Чтобы запустить IPython, нажмите клавишу <RETURN>)
```

После нажатия на клавишу Return IPython выведет следующий текст:

```
jmjones@dinkgutsy:stable-dev$ python ipython.py
Python 2.5.1 (r251:54863, Mar 7 2008, 03:39:23)
Type "copyright", "credits" or "license" for more information.
(Введите "copyright", "credits" или "license", чтобы получить
дополнительную информацию)

IPython 0.8.3.bzr.r96 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
             (Введение и обзор возможностей IPython)
%quickref -> Quick reference.
             (Краткий справочник)
help      -> Python's own help system.
             (Собственная справочная система Python)
object?   -> Details about 'object'. ?object also works, ?? prints more.
             (Подробности об 'object'. ?object также допустимо,
             ?? выведет более подробную информацию)

In [1]:
```

Взаимодействие с IPython

Обычно, когда сталкиваешься с новой командной строкой, в первый момент чувствуешь некоторую беспомощность. Совершенно непонятно – что делать дальше. Помните, как вы в первый раз вошли в систему UNIX и столкнулись с командной оболочкой (ba|k|c|z)sh? Раз уж вы читаете эту книгу, мы полагаем, что у вас уже имеется некоторый опыт работы с командной оболочкой в операционной системе UNIX. Если это действительно так, тогда обрести навыки работы в IPython будет совсем просто.

Одна из причин, по которым непонятно, как действовать делать в оболочке IPython, состоит в том, что она практически не ограничивает

ваших действий. Поэтому здесь уместнее думать о том, что бы вы хотели сделать. В командной оболочке IPython вам доступны все функциональные возможности языка Python. Плюс несколько «магических» функций IPython. Вы с легкостью можете запустить любую команду UNIX в оболочке IPython и сохранить вывод в переменной Python. Следующие примеры демонстрируют, что можно ожидать от IPython с настройками по умолчанию.

Ниже приводится пример выполнения некоторых простых операций присваивания:

```
In [1]: a = 1

In [2]: b = 2

In [3]: c = 3
```

Пока не видно существенных отличий от стандартной командной оболочки интерпретатора Python, если ввести в ней те же инструкции. Здесь мы просто присвоили значения 1, 2 и 3 переменным a, b и c, соответственно. Самое большое отличие между IPython и стандартной оболочкой Python, которое можно наблюдать здесь, состоит в том, что оболочка IPython выводит порядковый номер строки приглашения к вводу.

Теперь, когда у нас имеется несколько переменных (a, b и c), которым были присвоены значения (1, 2 и 3, соответственно), можно посмотреть, какие именно значения они содержат:

```
In [4]: print a
1

In [5]: print b
2

In [6]: print c
3
```

Конечно, это надуманный пример, потому что мы только что ввели эти значения, и можно было прокрутить назад окно, чтобы их увидеть. Вывод значения каждой переменной потребовал ввода с клавиатуры на шесть символов больше, чем это действительно необходимо. Ниже приводится более краткий способ отображения значений переменных:

```
In [7]: a
Out[7]: 1

In [8]: b
Out[8]: 2

In [9]: c
Out[9]: 3
```

Несмотря на то, что результаты этих двух способов выглядят практически одинаковыми, тем не менее, здесь имеются некоторые разли-

чия. Инструкция print использует «неофициальное» строковое представление, тогда как при использовании одного только имени переменной выводится «официальное» строковое представление. Обычно разница между этим двумя представлениями более заметна при работе не со встроенными, а с собственными классами. Ниже приводится пример различий между этими двумя строковыми представлениями:

```
In [10]: class DoubleRep(object):
   ....:     def __str__(self):
   ....:         return "Hi, I'm a __str__"
   ....:     def __repr__(self):
   ....:         return "Hi, I'm a __repr__"
   ....:
   ....:

In [11]: dr = DoubleRep()

In [12]: print dr
Hi, I'm a __str__

In [13]: dr
Out[13]: Hi, I'm a __repr__
```

Здесь с целью демонстрации различий между «официальным» и «неофициальным» строковыми представлениями объекта создается класс DoubleRep, обладающий двумя методами – __str__ и __repr__. Специальный метод __str__ объекта вызывается, когда требуется получить его «неофициальное» строковое представление. Специальный метод __repr__ объекта вызывается, когда требуется получить его «официальное» строковое представление. После создания экземпляра класса DoubleRep и присваивания его в качестве значения переменной dr мы выводим значение dr с помощью инструкции print. Вызывается метод __str__. Затем в строке приглашения к вводу мы просто вводим имя переменной, в результате чего вызывается метод __repr__. Таким образом, когда мы просто вводим имя переменной, оболочка IPython выводит ее «официальное» строковое представление. Когда мы используем инструкцию print, то получаем «неофициальное» строковое представление. Вообще в языке Python метод __str__ вызывается, когда происходит обращение к функции str(obj), которой передается желаемый объект, или когда он используется в строке форматирования, такой как эта: "%s" % obj. Когда происходит обращение к функции repr(obj) или используется строка форматирования, такая как "%r" % obj, вызывается метод __repr__.

Как бы то ни было, такой характер поведения не является особенностью исключительно оболочки IPython. Это особенность поведения интерпретатора Python. Ниже приводится тот же самый пример использования класса DoubleRep в стандартной интерактивной оболочке Python:

```
>>> class DoubleRep(object):
...     def __str__(self):
```

```
...         return "Hi, I'm a __str__"
...     def __repr__(self):
...         return "Hi, I'm a __repr__"
...
>>>
>>> dr = DoubleRep()
>>> print dr
Hi, I'm a __str__
>>> dr
Hi, I'm a __repr__
```

Вероятно, вы обратили внимание, что строки приглашения к вводу в стандартной оболочке Python и в оболочке IPython отличаются друг от друга. Строка приглашения в стандартной оболочке Python состоит из трех символов «больше» (>>>), тогда как в IPython приглашение содержит слово «In», за которым следует число в квадратных скобках и двоеточие (In [1]:) . Оказывается, оболочка IPython запоминает команды, которые вводились, и сохраняет их в списке с именем In. Так, после присваивания значений 1, 2 и 3 переменным a, b и c в предыдущем примере содержимое списка In выглядит следующим образом:

```
In [4]: print In
['\n', u'a = 1\n', u'b = 2\n', u'c = 3\n', u'print In\n']
```

Формат вывода результатов в IPython также отличается от вывода в стандартной оболочке Python. Создается впечатление, что оболочка IPython по-разному выводит значения в инструкции print и вычисленные. В действительности же оболочка IPython не делает никаких различий между этими двумя типами. Просто вызовы инструкций print являются побочным эффектом вычислений, поэтому оболочка IPython не видит их и не может их перехватить. Эти побочные проявления инструкции print отражаются только на стандартном потоке вывода stdout, куда передаются результаты запроса. Однако в ходе выполнения программного кода пользователя IPython оболочка контролирует возвращаемые значения. Если возвращаемое значение не равно None, оно выводится в строке с подсказкой Out [число]:.

В стандартной оболочке Python различия между этими двумя способами вывода вообще не видны. Если инструкция, введенная в строке приглашения IPython, вычисляет некоторое значение, отличное от None, оболочка выведет его в строке, которая начинается со слова Out, за которым следует число в квадратных скобках, символ двоеточия и значение, вычисленное инструкцией (например, Out[1]: 1). В следующем примере показано, как в оболочке IPython выполняется присваивание целочисленного значения переменной, как отображается значение переменной в результате ее оценки и как выводится значение этой же переменной с помощью инструкции print. Сначала в оболочке IPython:

```
In [1]: a = 1

In [2]: a
```

```
Out[2]: 1

In [3]: print a
1

In [4]:
```

А теперь в стандартной оболочке Python:

```
>>> a = 1
>>> a
1
>>> print a
1
>>>
```

В действительности нет никаких различий между тем, как в оболочках IPython и Python выполняется присваивание. Обе оболочки немедленно выводят приглашение к вводу. Но «официальное» строковое представление переменной в оболочке IPython и в стандартной оболочке Python отображается по-разному. В оболочке IPython выводится строка-подсказка `Out`, тогда как в оболочке Python просто выводится значение переменной. В случае же использования инструкции `print` никаких различий не наблюдается – в обеих оболочках выводится только значение.

Наличие подсказок `In [некоторое число]:` и `Out [некоторое число]:` может вызвать вопрос: имеются ли какие-нибудь более глубокие различия между IPython и стандартной оболочкой Python, или они носят исключительно косметический характер. Определенно, различия намного глубже. То есть видимые отличия от стандартной оболочки Python обусловлены функциональными возможностями оболочки IPython.

В оболочке IPython имеются две встроенные переменные, о существовании вам необходимо знать. Это переменные с именами `In` и `Out`. Первая из них представляет объект списка, в котором сохраняются введенные команды, а вторая – объект словаря. Вот что сообщает встроенная функция `type` о каждой из них:

```
In [1]: type(In)
Out[1]: <class 'IPython.iplib.InputList'>

In [2]: type(Out)
Out[2]: <type 'dict'>
```

Когда вы начнете пользоваться переменными `In` и `Out`, эти различия между ними приобретут особое значение.

Итак, что же хранится в этих переменных?

```
In [3]: print In
['\n', u'type(In)\n', u'type(Out)\n', u'print In\n']

In [4]: print Out
{1: <class 'IPython.iplib.InputList'>, 2: <type 'dict'>}
```

Как и следовало ожидать, переменные In и Out содержат, соответственно, отличные от None ввод команд и выражений и результаты выполнения инструкций и выражений . Поскольку каждая строка непременно содержит некоторый ввод, определенно имеет смысл сохранять введенные команды в виде такой структуры, как список. Но сохранение вывода в виде списка может привести к появлению элементов, которые содержат только значение None. Поэтому, т. к. не каждая введенная команда возвращает значение, отличное от None, есть смысл сохранять вывод в такой структуре данных, как словарь.

Дополнение

Другая невероятно полезная особенность IPython – функция дополнения, привязанная к клавише табуляции. Стандартная оболочка Python также обладает функцией дополнения – при условии, что интерпретатор скомпилирован с поддержкой библиотеки readline, но для ее активации необходимо выполнить следующие действия:

```
>>> import rlcompleter, readline
>>> readline.parse_and_bind('tab: complete')
```

Это позволит вам выполнять такие манипуляции, как показано ниже:

```
>>> import os
>>> os.lis<TAB>
>>> os.listdir
>>> os.li<TAB><TAB>
os.linesep  os.link     os.listdir
```

После импортирования модулей rlcompleter и readline и настройки параметра дополнения в модуле readline мы оказались в состоянии после импортирования модуля os ввести os.lis, нажать клавишу Tab один раз и получить дополнение до os.listdir. Точно так же, после ввода os.li мы получили список возможных вариантов дополнения, нажав клавишу Tab дважды.

Ту же самую функциональность можно получить в оболочке IPython, причем для этого не требуется выполнять подготовительных операций. То есть данная возможность в стандартной оболочке Python присутствует, а в IPython она активирована по умолчанию. Ниже приводится предыдущий пример, выполненный в оболочке IPython:

```
In [1]: import os

In [2]: os.lis<TAB>
In [2]: os.listdir
In [2]: os.li<TAB>
os.linesep  os.link    os.listdir
```

Обратите внимание: в последней части примера нам пришлось нажать клавишу табуляции всего один раз.

Этот пример всего лишь демонстрирует возможность поиска и дополнения атрибутов в оболочке IPython, но IPython, что может оказаться более привлекательным, может дополнять и имена модулей в инструкции импортирования. Откройте новую оболочку IPython, чтобы можно было увидеть, как IPython помогает отыскивать импортируемый модуль:

```
In [1]: import o
opcode       operator   optparse   os      os2emxpath ossaudiodev

In [1]: import xm
xml        xmllib     xmlrpclib
```

Обратите внимание, что все предлагаемые варианты дополнения являются именами модулей, то есть это уже не случайное совпадение. Это функциональная особенность.

В оболочке IPython есть два варианта дополнения: «дополнение» и «меню с дополнениями». При использовании первого варианта текущее «слово» дополняется по мере возможности и затем предлагается перечень альтернатив, при втором варианте слово дополняется полностью до одной из альтернатив, а каждое последующее нажатие клавиши Tab предоставляет трансформацию слова до следующей альтернативы. По умолчанию в оболочке IPython используется вариант «дополнение». К вопросам настройки IPython мы подойдем очень скоро.

Специальная функция редактирования

Последняя тема, касающаяся ввода-вывода, которую мы затронем, – это специальная функция `edit`. (Подробнее об специальных функциях мы поговорим в следующем разделе.) Взаимодействие пользователя с оболочкой, основанное на вводе строк, имеет огромное, но ограниченное значение. Так как это утверждение выглядит неоднозначным, попробуем развернуть его. Возможность ввода команд по одной строке за раз очень удобна. Вы вводите команду, а оболочка выполняет ее, причем иногда приходится ждать некоторое время, пока команда будет выполнена, а после этого вы вводите следующую команду.

В такой цикличности работы нет ничего плохого. В действительности такой способ взаимодействия достаточно эффективен. Но иногда возникает необходимость ввести сразу целый блок строк. Было бы неплохо иметь возможность использовать для этого предпочитаемый текстовый редактор, хотя аналогичную возможность предоставляет поддержка `readline` в IPython. Мы уже знаем, как использовать текстовый редактор для создания модулей на языке Python, но это не совсем то, что мы имеем в виду. Мы подразумеваем некоторый компромисс между строчно-ориентированным способом ввода и способом ввода в текстовом редакторе, который обеспечивает возможность передавать командной оболочке целые блоки строк с командами. Если можно сказать, что добавление поддержки возможности работы с блоками строк

совсем нелишне, это значит, что строчно-ориентированный интерфейс имеет некоторые ограничения. Т. е. можно сказать, что строчно-ориентированный интерфейс исключительно удобен, и в то же время имеет некоторые ограничения.

Специальная функция `edit` как раз и представляет собой упомянутый выше компромисс между строчно-ориентированным способом ввода в оболочке Python и способом ввода с привлечением текстового редактора. Преимущества такого компромисса состоят в том, что вы получаете в свои руки мощные возможности обоих способов редактирования. В вашем распоряжении имеются все преимущества вашего любимого текстового редактора. Вы легко можете редактировать блоки программного кода и изменять строки в пределах циклов, методов или функций. Плюс к этому вы не лишаетесь простоты и гибкости непосредственного взаимодействия с оболочкой. Комбинирование этих двух подходов еще больше усиливает их положительные стороны. Вы получаете возможность управлять своим рабочим окружением непосредственно из оболочки, вы можете приостанавливать работу, редактировать и выполнять программный код из текстового редактора. При возобновлении работы в оболочке вам будут доступны все изменения, выполненные в текстовом редакторе.

Настройка IPython

Последняя из «основ», о которой вам следует знать, – это порядок настройки IPython. Если вы не указывали иное местоположение при первом запуске оболочки IPython, она создаст каталог *.ipython* в вашем домашнем каталоге. Внутри каталога *.ipython* имеется файл с именем *ipy_user_conf.py*. Это обычный конфигурационный файл пользователя, в котором хранятся настройки, оформленные в виде синтаксических конструкций на языке Python. Конфигурационный файл хранит большое разнообразие элементов, позволяющих настраивать внешний вид и функциональные возможности IPython под себя. Например, имеется возможность выбрать цветовую гамму для оболочки, определить компоненты строки приглашения и выбрать текстовый редактор, который автоматически будет использоваться функцией `%edit` для ввода текста. Мы не будем углубляться в пояснения. Просто знайте, что такой конфигурационный файл существует и он стоит того, чтобы ознакомиться с его содержимым, – возможно, в нем вы найдете некоторые параметры, которые потребуется изменить.

Справка по специальным функциям

Как мы уже говорили, оболочка IPython обладает весьма широкими возможностями. Такая широта обусловлена наличием просто огромного числа встроенных специальных функций. Так что же такое специальная функция? В документации к IPython говорится:

Оболочка IPython рассматривает любую строку, начинающуюся с символа %, как вызов «специальной» функции. Эти функции позволяют управлять поведением самой оболочки IPython и добавляют ряд особенностей для работы с системой. Все имена специальных функций начинаются с символа %, при этом параметры передаются без использования круглых скобок или кавычек.

Пример: выполнение команды `'%cd mydir'` (без кавычек) изменит рабочий каталог на «mydir», если таковой существует.

Просмотреть и разобраться в этом многообразии дополнительных возможностей вам помогут две «специальные» функции. Первая специальная справочная функция, которую мы рассмотрим, – это функция `lsmagic`. Функция `lsmagic` выводит список всех «специальных» функций. Ниже приводится результат работы функции `lsmagic`:

```
In [1]: lsmagic
Available magic functions:
%Exit %Pprint %Quit %alias %autocall %autoindent %automagic %bg
%bookmark %cd %clear %color_info %colors %cpaste %debug %dhist %dirs
%doctest_mode %ed %edit %env %exit %hist %history %logoff %logon
%logstart %logstate %logstop %lsmagic %macro %magic %p %page %pdb
%pdef %pdoc %pfile %pinfo %popd %profile %prun %psearch %psource
%pushd %pwd %pycat %quickref %quit %r %rehash %rehashx %rep %reset
%run %runlog %save %sc %store %sx %system_verbose %time %timeit
%unalias %upgrade %who %who_ls %whos %xmode

Automagic is ON, % prefix NOT needed for magic functions.
(Автоматически ВКЛЮЧЕННЫЕ специальные функции, префикс %
для них НЕ требуется)
```

Как видите, существует огромное число доступных для вас специальных функций. Фактически, к моменту написания этих строк, существовало 69 специальных функций. Вы могли бы счесть более удобным получить список специальных функций следующим способом:

```
In [2]: %<TAB>
%Exit           %debug         %logstop    %psearch     %save
%Pprint         %dhist         %lsmagic    %psource     %sc
%Quit           %dirs          %macro      %pushd       %store
%alias          %doctest_mode  %magic      %pwd         %sx
%autocall       %ed            %p          %pycat       %system_verbose
%autoindent     %edit          %page       %quickref    %time
%automagic      %env           %pdb        %quit        %timeit
%bg             %exit          %pdef       %r           %unalias
%bookmark       %hist          %pdoc       %rehash      %upgrade
%cd             %history       %pfile      %rehashx     %who
%clear          %logoff        %pinfo      %rep         %who_ls
%color_info     %logon         %popd       %reset       %whos
%colors         %logstart      %profile    %run         %xmode
%cpaste         %logstate      %prun       %runlog
```

Ввод последовательности %-TAB в результате дает отформатированный список 69 специальных функций. Одним словом, функция lsmagic и комбинация %-TAB позволят вам быстро получить список всех имеющихся специальных функций, когда вы ищете что-то определенное или чтобы ознакомиться с тем, что вам доступно. Но список без описания не в состоянии помочь вам понять, для чего предназначена каждая функция.

Здесь к вам на помощь придет другая специальная справочная функция. Эта функция называется magic. Функция magic позволяет получить справочное описание всех специальных функций, встроенных в оболочку IPython. В справочную информацию включаются имя функции, порядок ее использования (область применения) и описание принципа действия функции. Ниже приводится описание функции page:

```
%page:
    Pretty print the object and display it through a pager.
    (Форматирует объект и отображает его с помощью программы
    постраничного просмотра)

    %page [options] OBJECT

    If no object is given, use _ (last output).
    (Если объект не указан, используется _ (последний введенный))

    Options:
    (Параметры)

    -r: page str(object), don't pretty-print it.
    (-r: page str(object), вывод информации в неформатированном виде)
```

В зависимости от используемой программы постраничного просмотра вы можете выполнять поиск и прокручивать результаты работы функции magic. Это может пригодиться, если вы знаете, что искать, чтобы перейти сразу к нужной странице вместо того, чтобы прокручивать описание к нужному месту. Описания функций упорядочены по алфавиту, что поможет вам быстро отыскать нужную функцию.

Можно также использовать и другой метод получения справочной информации, с которым мы познакомимся ниже в этой главе. Если ввести имя специальной функции и знак вопроса после нее (?), вы получите практически ту же самую информацию, что и с помощью функции %magic. Ниже приводится результат выполнения команды %page ?:

```
In [1]: %page ?
Type:           Magic function
Base Class:     <type 'instancemethod'>
String Form:    <bound method InteractiveShell.magic_page of
                <IPython.iplib.InteractiveShell object at 0x2ac5429b8a10>>
Namespace:      IPython internal
File:           /home/jmjones/local/python/psa/lib/python2.5/
                site-packages/IPython/Magic.py
Definition:     %page(self, parameter_s='')
```

```
Docstring:
    Pretty print the object and display it through a pager.

    %page [options] OBJECT

    If no object is given, use _ (last output).

    Options:

    -r: page str(object), don't pretty-print it.
```

И, наконец, еще одна справочная функция IPython, которая выводит сводный отчет об использовании различных возможностей, а также информацию о самих специальных функциях. Если в строке приглашения IPython ввести команду `%quickref`, вы получите справочник, который начинается со следующих строк:

```
IPython -- An enhanced Interactive Python - Quick Reference Card
(IPython – Расширенная интерактивная оболочка Python – краткий справочник)
================================================================

obj?, obj??      : Get help, or more help for object (also works as
                   ?obj, ??obj).
                  (Справка или подробная справка об объекте (допускается также
                 ?obj, ??obj).)
?foo.*abc*       : List names in 'foo' containing 'abc' in them.
                   (список имен в 'foo', содержащих 'abc')
%magic           : Information about IPython's 'magic' % functions.
                   (Информация о специальных функциях % IPython)

Magic functions are prefixed by %, and typically take their arguments without
parentheses, quotes or even commas for convenience.
(Имена специальных функций начинаются с % и обычно принимают аргументы без
использования скобок, кавычек и даже запятых)

Example magic function calls:
(Примеры вызова специальных функций)

%alias d ls -F   : 'd' is now an alias for 'ls -F'
                  (теперь 'd' – псевдоним для 'ls –F')
alias d ls -F    : Works if 'alias' not a python name
                  (Допустимо, если alias не является именем объекта Python)
alist = %alias   : Get list of aliases to 'alist'
                  (Записывает список псевдонимов в переменную 'alist')
cd /usr/share    : Obvious. cd -<tab> to choose from visited dirs.
                 (Очевидно. cd -<tab> для выбора из посещавшихся каталогов)
%cd??            : See help AND source for magic %cd
                 (См. справку И исходные тексты для специальной функции %cd)

System commands:

!cp a.txt b/     : System command escape, calls os.system()
                  (Экранирование системных команд, вызывается os.system())
cp a.txt b/      : after %rehashx, most system commands work without !
                 (после %rehashx, большинство системных команд работают без !)
cp ${f}.txt $bar : Variable expansion in magics and system commands
```

```
                (Подстановка имен переменных в специальных функциях
                и в системных командах)
files = !ls /usr : Capture sytem command output
                (Захватывает вывод системной команды)
files.s, files.l, files.n: "a b c", ['a','b','c'], 'a\nb\nc'
```

и заканчивается следующими строками:

```
%time:
    Time execution of a Python statement or expression.
    (Время выполнения инструкции или вычисления выражения)
%timeit:
    Time execution of a Python statement or expression
    (Время выполнения инструкции или вычисления выражения)
%unalias:
    Remove an alias
    (Удаляет псевдоним)
%upgrade:
    Upgrade your IPython installation
    (Обновляет версию IPython)
%who:
    Print all interactive variables, with some minimal formatting.
    (Выводит все интерактивные переменные с минимальным форматированием)
%who_ls:
    Return a sorted list of all interactive variables.
    (Возвращает отсортированный список всех интерактивных переменных)
%whos:
    Like %who, but gives some extra information about each variable.
    (Подобна функции %who, но выводит дополнительные сведения
     о каждой переменной)
%xmode:
    Switch modes for the exception handlers.
    (Переключает режим обработки исключений)
```

В самом начале вывода, получаемого от функции `%quickref`, приводится справочная информация о различных функциональных возможностях оболочки IPython. Остальная часть справочника `%quickref` представляет собой краткое описание всех специальных функций. Это краткое описание включает в себя первую строку из полной справки по каждой специальной функции. Например, ниже приводится полное описание функции `%who`:

```
In [1]: %who ?
Type:           Magic function
Base Class:     <type 'instancemethod'>
String Form:    <bound method InteractiveShell.magic_who of
                <IPython.iplib.InteractiveShell object at 0x2ac9f449da10>>
Namespace:      IPython internal
File:           /home/jmjones/local/python/psa/lib/python2.5/
                site-packages/IPython/Magic.py
Definition:     who(self, parameter_s='')
Docstring:
```

```
Print all interactive variables, with some minimal formatting.
(Выводит все интерактивные переменные с минимальным форматированием)

If any arguments are given, only variables whose type matches one of
these are printed. For example:
(Если указан какой-либо параметр, будут выведены переменные только
соответствующего типа. Например:)

    %who function str

will only list functions and strings, excluding all other types of
variables. To find the proper type names, simply use type(var) at
a command line to see how python prints type names. For example:
(выведет только имена функций и строковых переменных, исключая переменные
любых других типов. Чтобы отыскать требуемое имя типа, просто используйте
команду type(var), которая вернет имя типа в языке Python. Например:)

    In [1]: type('hello')
    Out[1]: <type 'str'>

indicates that the type name for strings is 'str'.
(указывает, что строки принадлежат к типу с именем 'str')

%who always excludes executed names loaded through your configuration
file and things which are internal to IPython.
(%who всегда исключает выполняемые имена, загруженные
из конфигурационного файла, и наименования,
являющиеся внутренними сущностями IPython.)

This is deliberate, as typically you may load many modules and the
purpose of %who is to show you only what you've manually defined.
(Сделано это преднамеренно, так как может быть загружено множество
модулей, а назначение функции %who состоит в том, чтобы показывать
только имена, определенные вручную.)
```

Справочная информация о функции `%who`, присутствующая в выводе функции `%quickref`, полностью идентична первой строке в разделе `Docstring` в блоке информации, которая возвращается командой `%who ?`.

Командная оболочка UNIX

У работы в командной оболочке UNIX есть свои преимущества (из которых можно назвать унифицированный подход к решению проблем, богатый набор инструментов, достаточно краткий и простой синтаксис, стандартные потоки ввода-вывода, конвейеры и перенаправление), но для нас было бы просто замечательно добавить этому старому другу еще и возможности Python. Оболочка IPython обладает рядом особенностей, которые повышают ценность соединения этих двух оболочек.

alias

Первая особенность объединения оболочки Python/UNIX, которую мы рассмотрим, – это специальная функция `alias`. С помощью этой функ-

ции можно создавать сокращенные псевдонимы системных команд. Чтобы определить псевдоним, достаточно просто ввести имя функции `alias` и далее указать системную команду (с любыми допустимыми параметрами). Например:

```
In [1]: alias nss netstat -lptn

In [2]: nss
(Not all processes could be identified, non-owned process info
 will not be shown, you would have to be root to see it all.)
(Не все процессы могут быть опознаны, информация о процессах, которыми
вы не владеете, отображаться не будет, вы должны иметь привилегии
пользователя root, чтобы увидеть все процессы.)
Active Internet connections (only servers)
(Активные соединения с Интернетом (только серверы)))
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp        0      0 0.0.0.0:80              0.0.0.0:*               LISTEN
tcp        0      0 127.0.0.1:631           0.0.0.0:*               LISTEN
```

Существует несколько способов передать дополнительные данные на вход псевдонима. Один из них – пассивный подход. Если все, что требуется передать псевдониму, допустимо смешивать в одну кучу, такой подход может оказаться полезным. Например, если с помощью утилиты `grep` из результатов команды `netstat`, показанных выше, необходимо отобрать только те, где номер порта равен 80, можно выполнить такую команду:

```
In [3]: nss | grep 80
(Not all processes could be identified, non-owned process info
 will not be shown, you would have to be root to see it all.)
tcp        0      0 0.0.0.0:80              0.0.0.0:*               LISTEN
```

Такой подход не годится, когда требуется передать дополнительные параметры, но для подобных случаев вполне подойдет.

Другой способ – активный подход. Он очень напоминает пассивный подход за исключением того, что наряду с неявными параметрами вы явно обрабатываете все последующие аргументы. Ниже приводится пример, демонстрирующий обработку всех дополнительных параметров как единой группы:

```
In [1]: alias achoo echo "|%l|"

In [2]: achoo
||

In [3]: achoo these are args
|these are args|
```

Здесь используется синтаксическая конструкция `%l` (знак процента, за которым следует символ «l»), которая вставляет оставшуюся часть командной строки в псевдоним. В реальной жизни такой прием, скорее

всего, использовался бы для вставки остальной части строки куда-нибудь в середину команды, для которой создается псевдоним.

И вот вам пример пассивного подхода, переделанный так, чтобы явно обрабатывать все дополнительные аргументы:

```
In [1]: alias nss netstat -lptn %l

In [2]: nss
(Not all processes could be identified, non-owned process info
 will not be shown, you would have to be root to see it all.)
Active Internet connections (only servers)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp        0      0 0.0.0.0:80              0.0.0.0:*               LISTEN
tcp        0      0 127.0.0.1:631           0.0.0.0:*               LISTEN

In [3]: nss | grep 80
(Not all processes could be identified, non-owned process info
 will not be shown, you would have to be root to see it all.)
tcp        0      0 0.0.0.0:80              0.0.0.0:*               LISTEN
```

В действительности, в этом примере не требовалось добавлять конструкцию `%l`. Если ее опустить, результат не изменится.

Для вставки параметров в середину командной строки можно было бы использовать параметр подстановки строки `%s`. Следующий пример демонстрирует, как выполняется обработка параметров:

```
In [1]: alias achoo echo first: "|%s|", second: "|%s|"

In [2]: achoo foo bar
first: |foo|, second: |bar|
```

Однако здесь может возникнуть проблема. Если псевдониму, который ожидает получить два параметра, передать только один, можно ждать появление ошибки:

```
In [3]: achoo foo
ERROR: Alias <achoo> requires 2 arguments, 1 given.
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
```

С другой стороны, вполне безопасно можно передавать большее число параметров:

```
In [4]: achoo foo bar bam
first: |foo|, second: |bar| bam
```

Параметры `foo` и `bar` были помещены в соответствующие позиции, а параметр `bam` просто был добавлен в конец, чего и следовало ожидать.

Сохранить псевдоним можно с помощью специальной функции `%store`, и ниже в этой главе будет показано, как это делается. Продолжая предыдущий пример, мы можем сохранить псевдоним `achoo`, чтобы при следующем запуске оболочки IPython его можно было использовать:

```
In [5]: store achoo
Alias stored: achoo (2, 'echo first: "|%s|", second: "|%s|"')

In [6]:
Do you really want to exit ([y]/n)?
(psa)jmjones@dinkgutsy:code$ ipython -nobanner

In [1]: achoo one two
first: |one|, second: |two|
```

Выполнение системных команд

Другой и, пожалуй, более простой способ выполнения системных команд заключается в использовании восклицательного знака (`!`) перед ними:

```
In [1]: !netstat -lptn
(Not all processes could be identified, non-owned process info
 will not be shown, you would have to be root to see it all.)
Active Internet connections (only servers)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp        0      0 0.0.0.0:80              0.0.0.0:*               LISTEN
tcp        0      0 127.0.0.1:631           0.0.0.0:*               LISTEN
```

Имеется возможность передавать системным командам значения переменных, при этом имена переменных должны начинаться со знака доллара (`$`). Например:

```
In [1]: user = 'jmjones'

In [2]: process = 'bash'

In [3]: !ps aux | grep $user | grep $process
jmjones   5967  0.0  0.4  21368  4344 pts/0    Ss+  Apr11   0:01 bash
jmjones   6008  0.0  0.4  21340  4304 pts/1    Ss   Apr11   0:02 bash
jmjones   8298  0.0  0.4  21296  4280 pts/2    Ss+  Apr11   0:04 bash
jmjones  10184  0.0  0.5  22644  5608 pts/3    Ss+  Apr11   0:01 bash
jmjones  12035  0.0  0.4  21260  4168 pts/15   Ss   Apr15   0:00 bash
jmjones  12943  0.0  0.4  21288  4268 pts/5    Ss   Apr11   0:01 bash
jmjones  15720  0.0  0.4  21360  4268 pts/17   Ss   02:37   0:00 bash
jmjones  18589  0.1  0.4  21356  4260 pts/4    Ss+  07:04   0:00 bash
jmjones  18661  0.0  0.0    320    16 pts/15   R+   07:06   0:00 grep bash
jmjones  27705  0.0  0.4  21384  4312 pts/7    Ss+  Apr12   0:01 bash
jmjones  32010  0.0  0.4  21252  4172 pts/6    Ss+  Apr12   0:00 bash
```

Здесь перечислены все сеансы работы с командной оболочкой `bash`, принадлежащие пользователю `jmjones`.

Ниже приводится пример сохранения результатов команды `!`:

```
In [4]: l = !ps aux | grep $user | grep $process

In [5]: l

Out[5]: SList (.p, .n, .l, .s, .grep(), .fields() available). Value:
0: jmjones   5967  0.0  0.4  21368  4344 pts/0    Ss+  Apr11   0:01 bash
```

```
1: jmjones   6008  0.0  0.4  21340  4304 pts/1   Ss   Apr11  0:02 bash
2: jmjones   8298  0.0  0.4  21296  4280 pts/2   Ss+  Apr11  0:04 bash
3: jmjones  10184  0.0  0.5  22644  5608 pts/3   Ss+  Apr11  0:01 bash
4: jmjones  12035  0.0  0.4  21260  4168 pts/15  Ss   Apr15  0:00 bash
5: jmjones  12943  0.0  0.4  21288  4268 pts/5   Ss   Apr11  0:01 bash
6: jmjones  15720  0.0  0.4  21360  4268 pts/17  Ss   02:37  0:00 bash
7: jmjones  18589  0.1  0.4  21356  4260 pts/4   Ss+  07:04  0:00 bash
8: jmjones  27705  0.0  0.4  21384  4312 pts/7   Ss+  Apr12  0:01 bash
9: jmjones  32010  0.0  0.4  21252  4172 pts/6   Ss+  Apr12  0:00 bash
```

Обратите внимание, что результат работы команды, сохраненный в переменной `l`, отличается от результата, полученного в предыдущем примере. Это потому, что переменная `l` содержит объект списка, тогда как в предыдущем примере демонстрировался обычный вывод команды. Подробнее объекты списков мы будем рассматривать в разделе «Обработка строк».

Альтернативой использованию символа `!` является использование комбинации `!!`. Эта комбинация обеспечивает возможность выполнять те же самые действия, что и `!`, за исключением того, что не позволяет сохранять результат в переменной. Но при ее использовании открывается возможность использовать конструкции `_` и `_[0-9]*`, которые будут обсуждаться ниже, в разделе «История результатов».

Использование `!` или `!!` перед системными командами определенно составляет меньше труда, чем создание псевдонимов, однако в одних случаях более предпочтительными могут оказаться псевдонимы, а в других – использование `!` или `!!`. Например, если предполагается, что некоторая команда будет использоваться постоянно, лучше создать для нее псевдоним. Для однократного или очень редкого использования лучше применять `!` или `!!`.

rehash

Существует еще один способ создания псевдонимов и/или выполнения системных команд из оболочки IPython, основанный на применении специальной функции `rehash` (выполняющей рехеширование). С технической точки зрения, она создает псевдонимы системных команд, но не совсем так, как это делали бы вы сами. Специальная функция `rehash` дополняет «таблицу псевдонимов» всем, что будет обнаружено в пути поиска файлов, определяемым переменной окружения `PATH`. Вы можете спросить: «Что это за таблица псевдонимов?». Когда создаются псевдонимы, оболочка IPython отображает имена псевдонимов на системные команды, с которыми они должны быть ассоциированы. Таблица псевдонимов как раз и описывает эти отображения.

Для рехеширования таблицы псевдонимов предпочтительнее использовать специальную функцию `rehashx`, а не `rehash`. Мы представим обе функции и покажем, как они работают, а затем опишем имеющиеся между ними отличия.

Во время работы оболочка IPython предоставляет в ваше распоряжение ряд переменных, таких как `In` и `Out`, с которыми мы уже встречались ранее. Одна из таких переменных называется `__IP`. Она представляет собой объект интерактивной оболочки. Атрибут этого объекта, с именем `alias_table`, ссылается на эту таблицу. Это именно то место, где хранятся отображения имен псевдонимов на системные команды. Мы можем просматривать эту таблицу точно так же, как любую другую переменную:

```
In [1]: __IP.alias_table

Out[1]:
{'cat': (0, 'cat'),
 'clear': (0, 'clear'),
 'cp': (0, 'cp -i'),
 'lc': (0, 'ls -F -o --color'),
 'ldir': (0, 'ls -F -o --color %l | grep /$'),
 'less': (0, 'less'),
 'lf': (0, 'ls -F -o --color %l | grep ^-'),
 'lk': (0, 'ls -F -o --color %l | grep ^l'),
 'll': (0, 'ls -lF'),
 'lrt': (0, 'ls -lart'),
 'ls': (0, 'ls -F'),
 'lx': (0, 'ls -F -o --color %l | grep ^-..x'),
 'mkdir': (0, 'mkdir'),
 'mv': (0, 'mv -i'),
 'rm': (0, 'rm -i'),
 'rmdir': (0, 'rmdir')}
```

Эта таблица выглядит как словарь:

```
In [2]: type(__IP.alias_table)

Out[2]: <type 'dict'>
```

Вид может оказаться обманчивым, но это не тот случай.

В настоящий момент в этом словаре имеется 16 элементов:

```
In [3]: len(__IP.alias_table)

Out[3]: 16
```

После вызова функции `rehash` объем словаря значительно увеличился:

```
In [4]: rehash

In [5]: len(__IP.alias_table)

Out[5]: 2314
```

Попробуем отыскать в нем то, чего не было прежде, но что должно было появиться, – теперь в словаре должна появиться утилита `transcode`:

```
In [6]: __IP.alias_table['transcode']

Out[6]: (0, 'transcode')
```

Когда вам встречается имя переменной или атрибута, начинающееся с двух символов подчеркивания (__), как правило, это означает, что программист не предполагал, что вы будете изменять содержимое этой переменной. Мы обратились к переменной __IP, но только для того, чтобы продемонстрировать вам внутреннее устройство. В случае необходимости мы могли бы воспользоваться официальным прикладным интерфейсом (API) IPython и обратиться к объекту _ip, доступному из командной строки оболочки IPython.

rehashx

Специальная функция rehashx по своему действию напоминает специальную функцию rehash, за исключением того, что при просмотре каталогов, перечисленных в переменной окружения PATH, она добавляет в таблицу псевдонимов только имена исполняемых файлов. Поэтому резонно предположить, что сразу после запуска оболочки IPython в результате работы функции rehashx таблица псевдонимов будет иметь тот же или меньший объем, как после запуска функции rehash:

```
In [1]: rehashx

In [2]: len(__IP.alias_table)

Out[2]: 2307
```

Интересно: после запуска функции rehashx размер таблицы псевдонимов оказался на семь элементов меньше, чем после вызова функции rehash. Ниже приводятся эти семь отличий:

```
In [3]: from sets import Set

In [4]: rehashx_set = Set(__IP.alias_table.keys())

In [5]: rehash

In [6]: rehash_set = Set(__IP.alias_table.keys())

In [7]: rehash_set - rehashx_set

Out[7]: Set(['fusermount', 'rmmod.modutils', 'modprobe.modutils', 'kallsyms',/
        'ksyms', 'lsmod.modutils', 'X11'])
```

И если поинтересоваться, почему, например, файл rmmod.modutils был отобран функцией rehash, но не был отобран функцией rehashx, можно обнаружить следующее:

```
jmjones@dinkgutsy:Music$ slocate rmmod.modutils
/sbin/rmmod.modutils
jmjones@dinkgutsy:Music$ ls -l /sbin/rmmod.modutils
lrwxrwxrwx 1 root root 15 2007-12-07 10:34 /sbin/rmmod.modutils ->
insmod.modutils
jmjones@dinkgutsy:Music$ ls -l /sbin/insmod.modutils
ls: /sbin/insmod.modutils: No such file or directory
```

Здесь видно, что `rmmod.utils` — это ссылка на `insmod.modutils`, но `insmod.modutils` отсутствует на диске.

cd

Если вам приходилось работать в стандартной оболочке Python, то, возможно, вы заметили, что в ней достаточно сложно определить имя каталога, в котором вы находитесь. Можно использовать функцию `os.chdir()`, чтобы перейти в нужный каталог, но это не очень удобно. Имя текущего каталога можно узнать с помощью функции `os.getcwd()`, но это тоже крайне неудобно. Поскольку в стандартной оболочке Python выполняются команды языка Python, это может выглядеть не такой большой проблемой, но при работе в оболочке IPython и наличии более простого доступа к системным командам достаточно важно иметь возможность более простого перемещения по каталогам.

Попробуйте воспользоваться специальной функцией `cd`. Вам кажется, что мы придаем этому большее значение, чем оно заслуживает: здесь нет ничего революционного и вполне можно обойтись без этой функции. Но только представьте, что ее нет. Жизнь без нее оказалась бы намного сложнее.

В оболочке IPython функция `cd` работает практически так же, как одноименная команда Bash. Типичный пример ее использования: `cd directory_name`. Этого вполне можно было ожидать из опыта работы в Bash. При вызове без аргументов функция `cd` выполняет переход в домашний каталог пользователя. Если после имени функции добавить пробел и дефис, `cd -`, она выполнит переход в предыдущий каталог. Функция `cd` может принимать три дополнительных аргумента, которые отсутствуют у одноименной команды в Bash.

Первый аргумент: `-q`, или `quiet`. Если этот аргумент не указать, IPython будет выводить имя каталога, куда был выполнен переход. В следующем примере демонстрируются способы изменения текущего каталога как с применением аргумента `–q`, так и без него:

```
In [1]: cd /tmp
/tmp

In [2]: pwd

Out[2]: '/tmp'

In [3]: cd -
/home/jmjones

In [4]: cd -q /tmp

In [5]: pwd

Out[5]: '/tmp'
```

Указание аргумента `-q` запрещает IPython вывод имени каталога */tmp*, в который был выполнен переход.

Еще одной особенностью функции `cd` в IPython является возможность перехода по определенным ранее закладкам. (Вскоре мы объясним, как они создаются.) В следующем примере показано, как выполнить переход в каталог по созданной ранее закладке:

```
In [1]: cd -b t
(bookmark:t) -> /tmp
/tmp
```

В этом примере предполагается, что ранее для каталога */tmp* была создана закладка с именем `t`. Формально синтаксис перехода в каталог по закладке имеет следующий вид: `cd -b bookmark_name`, но если закладка `bookmark_name` определена и в текущем каталоге отсутствует подкаталог *bookmark_name*, то ключ `-b` можно опустить – в этом случае оболочка IPython предполагает, что вы собираетесь выполнить переход по закладке.

Последняя дополнительная особенность, которую предлагает функция `cd` в оболочке IPython, заключается в возможности перейти в определенный каталог, присутствующий в списке ранее посещавшихся каталогов. Ниже приводится пример использования этого списка:

```
0: /home/jmjones
1: /home/jmjones/local/Videos
2: /home/jmjones/local/Music
3: /home/jmjones/local/downloads
4: /home/jmjones/local/Pictures
5: /home/jmjones/local/Projects
6: /home/jmjones/local/tmp
7: /tmp
8: /home/jmjones

In [2]: cd -6
/home/jmjones/local/tmp
```

В первой части примера приводится список ранее посещавшихся каталогов. Как его получить, мы совсем скоро расскажем. Затем следует вызов функции `cd`, которой передается числовой аргумент `-6`. Он сообщает оболочке IPython, что нам требуется перейти в каталог, который находится в списке под номером «6», то есть в каталог */home/jmjones/local/tmp*. И в последней строке оболочка сообщает, что теперь вы находитесь в каталоге */home/jmjones/local/tmp*.

bookmark

Мы только что продемонстрировали, как использовать закладки в функции `cd` для перехода в требуемый каталог. А теперь мы покажем, как создавать эти закладки и как ими управлять. Следует упомянуть, что закладки сохраняются между сеансами работы с оболочкой IPython. Если завершить работу с оболочкой, а затем вновь запустить

ее, закладки будут восстановлены. Создать закладку можно двумя способами. Ниже демонстрируется первый из них:

```
In [1]: cd /tmp
/tmp

In [2]: bookmark t
```

Выполнив команду `bookmark t` после перехода в каталог */tmp*, мы создали закладку с именем `t`, указывающую на каталог */tmp*. Второй способ создания закладки требует ввести более чем одно слово:

```
In [3]: bookmark muzak /home/jmjones/local/Music
```

Здесь была создана закладка с именем `muzak`, которая указывает на локальный каталог с музыкой. Первый аргумент – это имя закладки, а второй – имя каталога, на который ссылается закладка.

Получить список закладок, которых у нас к настоящему моменту всего две, можно с помощью параметра `-l`. Посмотрим, как выглядит список всех наших закладок:

```
In [4]: bookmark -l
Current bookmarks:
muzak -> /home/jmjones/local/Music
t     -> /tmp
```

Удалять закладки можно двумя способами: удалить сразу все закладки или только выбранную. В следующем примере создается новая закладка, затем она удаляется, а после этого удаляются все остальные закладки:

```
In [5]: bookmark ulb /usr/local/bin

In [6]: bookmark -l
Current bookmarks:
muzak -> /home/jmjones/local/Music
t     -> /tmp
ulb   -> /usr/local/bin

In [7]: bookmark -d ulb

In [8]: bookmark -l
Current bookmarks:
muzak -> /home/jmjones/local/Music
t     -> /tmp
```

Вместо команды `bookmark -l` можно использовать команду `cd -b`:

```
In [9]: cd -b<TAB>
muzak t
```

Нажав несколько раз клавишу Backspace, продолжим с того места, где остановились:

```
In [9]: bookmark -r
```

```
In [10]: bookmark -l
Current bookmarks:
```

В этом примере сначала была создана закладка с именем `ulb`, указывающая на каталог */usr/local/bin*. Затем она была удалена с помощью аргумента `-d bookmark_name` команды `bookmark`. В конце были удалены все закладки с помощью аргумента `-r`.

dhist

В примере использования функции `cd` был продемонстрирован список посещавшихся ранее каталогов. Этот список сохраняется не только в течение сеанса, но и между сеансами работы с оболочкой IPython. Ниже демонстрируется пример вызова функции `dhist` без аргументов:

```
In [1]: dhist
Directory history (kept in _dh)
0: /home/jmjones
1: /home/jmjones/local/Videos
2: /home/jmjones/local/Music
3: /home/jmjones/local/downloads
4: /home/jmjones/local/Pictures
5: /home/jmjones/local/Projects
6: /home/jmjones/local/tmp
7: /tmp
8: /home/jmjones
9: /home/jmjones/local/tmp
10: /tmp
```

Быстро получить доступ к этому списку можно с помощью команды `cd -<TAB>`, как показано ниже:

```
In [1]: cd -<TAB>
-00 [/home/jmjones]                -06 [/home/jmjones/local/tmp]
-01 [/home/jmjones/local/Videos]   -07 [/tmp]
-02 [/home/jmjones/local/Music]    -08 [/home/jmjones]
-03 [/home/jmjones/local/downloads] -09 [/home/jmjones/local/tmp]
-04 [/home/jmjones/local/Pictures] -10 [/tmp]
-05 [/home/jmjones/local/Projects]
```

Две дополнительных возможности функции `dhist` делают ее более гибкой, чем команда `cd -<TAB>`. В первом случае имеется возможность указать, сколько каталогов должно быть отображено. Например, чтобы указать, что требуется отобразить только пять последних посещавшихся каталогов, можно воспользоваться такой командой:

```
In [2]: dhist 5
Directory history (kept in _dh)
6: /home/jmjones/local/tmp
7: /tmp
8: /home/jmjones
9: /home/jmjones/local/tmp
10: /tmp
```

Во втором – определить диапазон элементов списка посещавшихся ранее каталогов. Например, чтобы просмотреть каталоги в списке с третьего по шестой, можно выполнить следующую команду:

```
In [3]: dhist 3 7
Directory history (kept in _dh)
3: /home/jmjones/local/downloads
4: /home/jmjones/local/Pictures
5: /home/jmjones/local/Projects
6: /home/jmjones/local/tmp
```

Обратите внимание: элемент списка с номером, соответствующим второму аргументу, не включается в вывод, поэтому второе число должно соответствовать порядковому номеру элемента списка, следующему непосредственно за последним каталогом, который требуется вывести.

pwd

Это простая функция, но она часто бывает необходима при навигации в дереве каталогов. Функция `pwd` выводит имя текущего каталога. Например:

```
In [1]: cd /tmp
/tmp

In [2]: pwd

Out[2]: '/tmp'
```

Подстановка переменных

Предыдущие особенности оболочки IPython определенно удобны и необходимы, но следующие три особенности доставят массу удовольствия искушенным пользователям. Первая из них – подстановка имен переменных. До настоящего момента мы использовали в командной оболочке только то, что имеет отношение к командной оболочке, а в Python – только то, что принадлежит языку Python. Но теперь мы попробуем соединить это вместе. То есть мы попробуем взять значение, которое было получено интерпретатором Python, и передать его командной оболочке:

```
In [1]: for i in range(10):
...:         !date > ${i}.txt
...:
...:

In [2]: ls
0.txt 1.txt 2.txt 3.txt 4.txt 5.txt 6.txt 7.txt 8.txt 9.txt

In [3]: !cat 0.txt
Sat Mar 8 07:40:05 EST 2008
```

Это достаточно надуманный пример. Едва ли вам потребуется создать 10 текстовых файлов, каждый из которых содержит дату. Но этот пример наглядно демонстрирует, как смешивать программный код на языке Python с программным кодом на языке командной оболочки. В этом примере выполняются итерации по списку, созданному функцией `range()`, каждый элемент которого поочередно сохраняется в переменной `i`. В каждой итерации с использованием нотации `!` вызывается системная утилита `date`. Обратите внимание, что синтаксис вызова `date` идентичен способу, который использовался бы, если бы мы определили переменную окружения `i`. При таком подходе производится вызов утилиты `date`, а ее вывод перенаправляется в файл с именем *{текущий элемент списка}.txt*. После этого в примере выводится список созданных файлов и даже содержимое одного из них, чтобы убедиться, что он содержит нечто, напоминающее дату.

В системную оболочку можно передавать любые значения, полученные в Python. Независимо от того, получены эти значения из базы данных, созданы в ходе вычислений, получены в результате обращения к службе XMLRPC или извлечены из текстового файла, вы можете получить их средствами языка Python и передать системной команде, применяя прием с использованием `!`.

Обработка строк

Другой невероятно мощной особенностью, которую предлагает оболочка IPython, является возможность обрабатывать строки, полученные от системных команд. Предположим, что нам необходимо получить идентификаторы всех процессов (PID), принадлежащих пользователю `jmjones`. Для этого можно было бы использовать следующую команду:

```
ps aux | awk '{if ($1 == "jmjones") print $2}'
```

Эта команда выглядит достаточно компактной и понятной. Но попробуем решить ту же самую задачу средствами IPython. Для начала получим вывод от команды `ps aux`:

```
In [1]: ps = !ps aux

In [2]:
```

Результат работы команды `ps aux` сохраняется в переменной `ps` в виде списка, элементами которого являются строки, полученные от системной команды. Под словами «в виде списка» в данном случае подразумевается, что переменная принадлежит к встроенному типу `list`, поэтому она поддерживает все методы, принадлежащие этому типу. Благодаря этому, если у вас имеется функция, которая ожидает получить список, вы можете передать ей полученный объект с результатами. Кроме того, помимо стандартных методов списка она поддерживает

еще пару весьма интересных методов и один атрибут, которые могут вам пригодиться. Только ради того, чтобы продемонстрировать эти «интересные методы», мы пока отложим задачу получения всех процессов, принадлежащих пользователю `jmjones`. Первый «интересный метод» – это метод `grep()`. Фактически он представляет собой обычный фильтр, который определяет, какие строки оставить, а какие отбросить. Чтобы узнать, имеются ли какие-нибудь строки, содержащие слово `lighthttp`, мы могли бы воспользоваться следующей командой:

```
In [2]: ps.grep('lighttpd')

Out[2]: SList (.p, .n, .l, .s, .grep(), .fields() available). Value:
0: www-data 4905 0.0 0.1........0:00 /usr/sbin/lighttpd -f /etc/lighttpd/l
```

Здесь мы вызвали метод `grep()` и передали ему регулярное выражение `'lighthttp'`. Запомните: регулярные выражения, которые передаются методу `grep()`, не чувствительны к регистру символов. В результате этого вызова метода `grep()` была получена строка, где было найдено соответствие регулярному выражению `'lighthttp'`. Чтобы получить все записи, за исключением тех, что соответствуют указанному регулярному выражению, мы могли бы использовать примерно такую команду:

```
In [3]: ps.grep('Mar07', prune=True)

Out[3]: SList (.p, .n, .l, .s, .grep(), .fields() available). Value:
0: USER       PID %CPU %MEM    VSZ    RSS TTY    STAT START  TIME COMMAND
1: jmjones  19301  0.0  0.4  21364   4272 pts/2  Ss+  03:58  0:00 bash
2: jmjones  21340  0.0  0.9  202484 10184 pts/3  Sl+  07:00  0:06 vim ipytho
3: jmjones  23024  0.0  1.1  81480  11600 pts/4  S+   08:58  0:00 /home/jmjo
4: jmjones  23025  0.0  0.0      0      0 pts/4  Z+   08:59  0:00 [sh] <defu
5: jmjones  23373  5.4  1.0  81160  11196 pts/0  R+   09:20  0:00 /home/jmjo
6: jmjones  23374  0.0  0.0   3908    532 pts/0  R+   09:20  0:00 /bin/sh -c
7: jmjones  23375  0.0  0.1  15024   1056 pts/0  R+   09:20  0:00 ps aux
```

Мы передали методу `grep()` регулярное выражение `'Mar07'` и обнаружили, что большинство процессов было запущено 7 марта, поэтому мы решили получить все процессы, которые были запущены *не* 7 марта. Чтобы исключить все записи, соответствующие регулярному выражению `'Mar07'`, мы добавили еще один аргумент: `prune=True`. Этот именованный аргумент сообщает оболочке IPython, что «любые записи, соответствующие указанному регулярному выражению, должны быть отброшены». И, как видите, в полученном списке нет ни одной записи, соответствующей регулярному выражению `'Mar07'`.

С методом `grep()` можно также использовать функции обратного вызова. Это означает, что метод `grep()` может принимать в виде аргумента функцию и вызывать ее. Он передает функции текущий элемент списка. Если функция возвращает `True` для этого элемента, он включается в итоговый набор. Например, мы могли бы получить содержимое каталога и оставить в нем только файлы или только каталоги:

```
In [1]: import os

In [2]: file_list = !ls

In [3]: file_list

Out[3]: SList (.p, .n, .l, .s, .grep(), .fields() available). Value:
0: ch01.xml
1: code
2: ipython.pdf
3: ipython.xml
```

Этот каталог содержит «файлы». Мы не можем сказать, какие из них действительно являются файлами, а какие каталогами, но если воспользоваться фильтром `os.path.isfile()`, мы сможем отобрать только те, которые являются файлами:

```
In [4]: file_list.grep(os.path.isfile)

Out[4]: SList (.p, .n, .l, .s, .grep(), .fields() available). Value:
0: ch01.xml
1: ipython.pdf
2: ipython.xml
```

В этом списке отсутствует «файл» *code*, следовательно, он вообще не является файлом. Попробуем отобрать каталоги:

```
In [5]: file_list.grep(os.path.isdir)

Out[5]: SList (.p, .n, .l, .s, .grep(), .fields() available). Value:
0: code
```

Теперь видно, что *code* действительно является каталогом. Другой интересный метод – это метод `fields()`. После (или даже до) того, как будет выполнена фильтрация набора результатов в соответствии с определенными требованиями, вы можете отобразить те поля, которые желательно было бы вывести. Вернемся к нашему предыдущему примеру, где выводились записи о процессах, запущенных не 7 марта, и выведем только информацию в полях `USER`, `PID` и `START`:

```
In [4]: ps.grep('Mar07', prune=True).fields(0, 1, 8)

Out[4]: SList (.p, .n, .l, .s, .grep(), .fields() available). Value:
0: USER PID START
1: jmjones 19301 03:58
2: jmjones 21340 07:00
3: jmjones 23024 08:58
4: jmjones 23025 08:59
5: jmjones 23373 09:20
6: jmjones 23374 09:20
7: jmjones 23375 09:20
```

Во-первых, обратите внимание, что метод `fields()` применяется к результатам, возвращаемым методом `grep()`. Это возможно потому, что метод `grep()` возвращает объект того же типа, что и объект `ps`. Метод

fields() так же возвращает объект того же типа, что и метод grep(). Благодаря этому мы смогли объединить в цепочку методы grep() и fields(). Теперь подробнее о том, что здесь происходит. Метод fields() принимает неопределенное число аргументов, которые, как предполагается, обозначают номера «колонок» в выводе, при этом предполагается, что колонки отделяются пробелами. Это очень похоже на то, как awk разбивает строки текста. В данном случае методу fields() предписывается вывести колонки с порядковыми номерами 0, 1 и 8. Эти номера соответствуют колонкам USER, PID и START.

Теперь вернемся к задаче отображения идентификаторов всех процессов (PID), принадлежащих пользователю jmjones:

```
In [5]: ps.fields(0, 1).grep('jmjones').fields(1)

Out[5]: SList (.p, .n, .l, .s, .grep(), .fields() available). Value:
0: 5385
1: 5388
2: 5423
3: 5425
4: 5429
5: 5431
6: 5437
7: 5440
8: 5444
<продолжение списка...>
```

В этом примере сначала отбираются только первые два столбца, 0 и 1, которые соответствуют колонкам USER и PID, соответственно. Затем из полученного списка, с помощью метода grep(), отбираются только те записи, которые соответствуют регулярному выражению 'jmjones'. И в заключение, из полученного набора выводится только второе поле с помощью вызова метода fields(1). (Не забывайте, что нумерация полей начинается с нуля.)

Последний элемент, имеющий отношение к обработке строк, из тех, которые нам хотелось бы продемонстрировать, – это атрибут s объекта, который позволяет получить непосредственный доступ к списку. Возможно, что результаты, которые дает сам объект, – не совсем то, что вам хотелось бы получить. Поэтому для передачи ваших данных системной командной оболочке используйте атрибут s списка:

```
In [6]: ps.fields(0, 1).grep('jmjones').fields(1).s

Out[6]: '5385 5388 5423 5425 5429 5431 5437 5440 5444 5452 5454 5457
5458 5468 5470 5478 5480 5483 5489 5562 5568 5593 5595 5597 5598 5618
5621 5623 5628 5632 5640 5740 5742 5808 5838 12707 12913 14391 14785
19301 21340 23024 23025 23373 23374 23375'
```

При обращении к атрибуту s возвращается обычная строка, содержащая идентификаторы процессов, разделенные пробелами, с которой можно работать средствами командной оболочки. При желании этот

список в виде строки можно было сохранить в переменной с именем pids и затем в оболочке IPython выполнить, например, такую команду: kill $pids. Но такая команда послала бы сигнал SIGTERM всем процессам, принадлежащим пользователю jmjones, что привело бы к завершению работы текстового редактора и сеанса IPython.

Ранее уже демонстрировалось, что та же самая задача может быть решена с помощью однострочного сценария на языке awk:

```
ps aux | awk '{if ($1 == "jmjones") print $2}'
```

Мы будем готовы добиться тех же результатов после того, как рассмотрим еще одну концепцию. Метод grep() принимает еще один необязательный параметр с именем field. Если параметр field определен, во время принятия решения о включении очередного элемента в результат критерий поиска будет применяться к указанному полю:

```
In [1]: ps = !ps aux

In [2]: ps.grep('jmjones', field=0)

Out[2]: SList (.p, .n, .l, .s, .grep(), .fields() available). Value:
0: jmjones   5361  0.0  0.1  46412  1828 ?        SL   Apr11
  0:00 /usr/bin/gnome-keyring-daemon -d
1: jmjones  5364  0.0  1.4  214948 14552 ?        Ssl  Apr11
  0:03 x-session-manager
....
53: jmjones 32425 0.0  0.0    3908   584 ?        S    Apr15
   0:00 /bin/sh /usr/lib/firefox/run-mozilla.
54: jmjones 32429 0.1  8.6  603780 88656 ?        Sl   Apr15
   2:38 /usr/lib/firefox/firefox-bin
```

В этом случае были отобраны требуемые строки, но они были получены целиком. Чтобы отобрать только идентификаторы процессов, можно предусмотреть следующее действие:

```
In [3]: ps.grep('jmjones', field=0).fields(1)

Out[3]: SList (.p, .n, .l, .s, .grep(), .fields() available). Value:
0: 5361
1: 5364
....
53: 32425
54: 32429
```

Теперь мы имеем средства достичь той же цели, что и фильтр на языке awk.

Профиль sh

Одно из понятий IPython, которое еще не было описано, – это профиль. Профиль – это набор конфигурационных данных, которые загружаются при запуске оболочки IPython. Имеется возможность соз-

давать произвольное число профилей для настройки IPython в зависимости от потребностей. Для вызова определенной конфигурации следует использовать ключ командной строки `-p` и указать имя желаемого профиля.

Профиль `sh` (или shell) – это один из встроенных профилей IPython. Профиль `sh` определяет значения некоторых конфигурационных параметров, в результате чего оболочка IPython становится более дружественной по отношению к системной командной оболочке. Приведем два примера параметров конфигурации, имеющих значения, отличные от значений в стандартном профиле IPython: параметр, задающий отображение текущего каталога в строке приглашения к вводу, и параметр, задающий рехеширование каталогов, перечисленных в переменной окружения `PATH`, что обеспечивает моментальный доступ ко всем исполняемым файлам, к которым он имеется, например, в оболочке Bash.

Помимо установки некоторых конфигурационных значений профиль `sh` активирует некоторые полезные расширения. Например, он активирует расширение `envpersist`. Расширение `envpersist` позволяет изменять различные переменные окружения и запоминать их значения в профиле `sh`, благодаря чему ликвидируется необходимость обновлять содержимое файла `.bash_profile` или `.bashrc`.

Ниже показано, как выглядит значение переменной `PATH`:

```
jmjones@dinkgutsy:tmp$ ipython -p sh
IPython 0.8.3.bzr.r96 [on Py 2.5.1]
[~/tmp]|2> import os
[~/tmp]|3> os.environ['PATH']
       <3> '/home/jmjones/local/python/psa/bin:
       /home/jmjones/apps/lb/bin:/home/jmjones/bin:
       /usr/local/sbin:/usr/local/bin:/usr/sbin:
       /usr/bin:/sbin:/bin:/usr/games'
```

Теперь добавим `:/appended` в конец значения переменной `PATH`:

```
[~/tmp]|4> env PATH+=:/appended
PATH after append = /home/jmjones/local/python/psa/bin:
/home/jmjones/apps/lb/bin:/home/jmjones/bin:
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:
/sbin:/bin:/usr/games:/appended
```

а `/prepended:` в начало:

```
[~/tmp]|5> env PATH-=/prepended:
PATH after prepend = /prepended:/home/jmjones/local/python/psa/bin:
/home/jmjones/apps/lb/bin:/home/jmjones/bin:/usr/local/sbin:
/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/appended
```

Теперь посмотрим содержимое переменной `PATH` с помощью `os.environ`:

```
[~/tmp]|6> os.environ['PATH']
```

```
        <6> '/prepended:/home/jmjones/local/python/psa/bin:
        /home/jmjones/apps/lb/bin:/home/jmjones/bin:
        /usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:
        /bin:/usr/games:/appended'
```

Закроем оболочку IPython:

```
[~/tmp]|7>
Do you really want to exit ([y]/n)?
jmjones@dinkgutsy:tmp$
```

Теперь откроем снова оболочку IPython, чтобы взглянуть на содержимое переменной `PATH`:

```
jmjones@dinkgutsy:tmp$ ipython -p sh
IPython 0.8.3.bzr.r96 [on Py 2.5.1]
[~/tmp]|2> import os
[~/tmp]|3> os.environ['PATH']
        <3> '/prepended:/home/jmjones/local/python/psa/bin:
        /home/jmjones/apps/lb/bin:/home/jmjones/bin:/usr/local/sbin:
        /usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/appended'
```

Как видите, добавленные значения остались на своих местах, и при этом нам не потребовалось изменять какие-либо конфигурационные сценарии. Значение переменной `PATH` было сохранено без нашего вмешательства. Теперь посмотрим, какие наши изменения переменных окружения сохраняются:

```
[~/tmp]|4> env -p
        <4> {'add': [('PATH', ':/appended')], 'pre': [('PATH', '/
prepended:')],
        'set': {}}
```

Мы можем удалить сохраняемые изменения значения переменной `PATH`:

```
[~/tmp]|5> env -d PATH
Forgot 'PATH' (for next session)
```

и проверить получившееся значение переменной `PATH`:

```
[~/tmp]|6> os.environ['PATH']
        <6>'/prepended:/home/jmjones/local/python/psa/bin:/home/jmjones/apps/
        lb/bin:/home/jmjones/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/
usr/
        bin:/sbin:/bin:/usr/games:/appended'
```

Хотя была дана команда удалить сохраняемые изменения для переменной `PATH`, они по-прежнему остаются на месте. Это означает лишь то, что оболочка IPython удалила указание на необходимость сохранения этих изменений. Обратите внимание, что процесс, запущенный с определенными значениями в переменной окружения, будет сохранять их, если не изменить их некоторым способом. При следующем запуске оболочки IPython окружение изменится:

```
[~/tmp]|7>
Do you really want to exit ([y]/n)?
jmjones@dinkgutsy:tmp$ ipython -p sh
IPython 0.8.3.bzr.r96 [on Py 2.5.1]
[~/tmp]|2> import os
[~/tmp]|3> os.environ['PATH']
       <3> '/home/jmjones/local/python/psa/bin:/home/jmjones/apps/lb/bin:
       /home/jmjones/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:
       /sbin:/bin:/usr/games'
```

Как и следовало ожидать, переменная `PATH` вернулась к значению, которое предшествовало тому моменту, как мы внесли в нее изменения.

Еще одна полезная особенность в профиле `sh` – это специальная функция `mglob`. Функция `mglob` имеет простой синтаксис для наиболее распространенных вариантов использования. Например, чтобы отыскать все файлы с расширением *.py* в проекте Django, можно было бы воспользоваться следующей командой:

```
[django/trunk]|3> mglob rec:*py
            <3> SList (.p, .n, .l, .s, .grep(), .fields() available). Value:
0: ./setup.py
1: ./examples/urls.py
2: ./examples/manage.py
3: ./examples/settings.py
4: ./examples/views.py
...
1103: ./django/conf/project_template/urls.py
1104: ./django/conf/project_template/manage.py
1105: ./django/conf/project_template/settings.py
1106: ./django/conf/project_template/__init__.py
1107: ./docs/conf.py
[django/trunk]|4>
```

Директива `rec` предписывает выполнить рекурсивный поиск по заданному вслед за ней шаблону. В данном случае шаблоном служит `*py`. Чтобы отобразить список всех каталогов в корневом каталоге проекта Django, можно было бы воспользоваться следующей командой:

```
[django/trunk]|3> mglob dir:*
             <3> SList (.p, .n, .l, .s, .grep(), .fields() available).
             Value:
             0: examples
             1: tests
             2: extras
             3: build
             4: django
             5: docs
             6: scripts
             </3>
```

Функция `mglob` возвращает объект списка, поэтому все, что в языке Python можно сделать со списком, можно сделать и с полученным списком каталогов.

Это была демонстрация лишь некоторых особенностей поведения профиля `sh`. Существуют другие особенности и параметры этого профиля, которые мы не рассматривали.

Сбор информации

IPython – это немножко больше, чем просто оболочка, в которой можно активно работать. Это еще и инструмент сбора разного рода информации о программном коде и объектах, с которыми приходится работать. Она обеспечивает такие возможности в добывании информации, что с легкостью может рассматриваться как инструмент исследователя. В этом разделе описывается ряд особенностей, которые помогут вам в сборе информации.

page

Если представление объекта, с которым приходится работать, не умещается на экране, можно попробовать воспользоваться специальной функцией `page`. Вы можете использовать функцию `page` для вывода объекта с помощью программы постраничного просмотра. Во многих системах в качестве такой программы по умолчанию используется утилита `less`, но вы можете использовать какую-нибудь другую программу. Ниже демонстрируется стандартный способ использования:

```
In [1]: p = !ps aux
==
['USER      PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND',
'root        1  0.0  0.1   5116  1964 ?        Ss   Mar07   0:00 /sbin/init',
< ... дальнейшие результаты обрезаны ... >

In [2]: page p
['USER      PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND',
'root        1  0.0  0.1   5116  1964 ?        Ss   Mar07   0:00 /sbin/init',
< ... дальнейшие результаты обрезаны ... >
```

Здесь в переменной `p` сохраняется результат выполнения системной команды `ps aux`. Затем вызывается функция `page`, которой передается объект с результатами. После этого функция `page` запускает программу `less`.

Функция `page` имеет один дополнительный параметр: `-r`. Этот параметр предписывает функции `page` передать программе постраничного просмотра неформатированное строковое представление объекта (результат вызова функции `str()`). Для нашего объекта со списком процессов это могло бы выглядеть так:

```
In [3]: page -r p
ilus-cd-burner/mapping-d', 'jmjones    5568 0.0 1.0 232004 10608 ?     S
Mar07   0:00 /usr/lib/gnome-applets/trashapplet --', 'jmjones   5593 0.0 0.9
188996 10076 ?     S    Mar07   0:00 /usr/lib/gnome-applets/battstat-apple',
'jmjones   5595 0.0 2.8 402148 29412 ?     S    Mar07 0:01 p
<... дальнейшие результаты обрезаны ...>
```

Такой неформатированный вывод выглядит почти нечитабельно. Мы рекомендуем начинать с форматированного вывода и работать уже с ним.

pdef

Специальная функция `pdef` выводит заголовки определений (сигнатуры функций) любых вызываемых объектов. В следующем примере мы создаем свою собственную функцию со строкой документирования и инструкцией `return`:

```
In [1]: def myfunc(a, b, c, d):
...:         '''return something by using a, b, c, d to do something'''
...:         return a, b, c, d
...:

In [2]: pdef myfunc
 myfunc(a, b, c, d)
```

Функция `pdef` проигнорировала строку документирования и инструкцию `return`, но вывела сигнатуру функции. Функцию `pdef` можно использовать с любой вызываемой функцией. Она может работать, даже когда исходный программный код недоступен, но при условии, что имеется доступ к файлу *.pyc* или к пакету.

pdoc

Функция `pdoc` выводит строку документирования переданной ей функции. Ниже мы передали `pdoc` функцию `myfunc`, которую передавали функции `pdef` в примере выше:

```
In [3]: pdoc myfunc
Class Docstring:
    return something by using a, b, c, d to do something
Calling Docstring:
    x.__call__(...) <==> x(...)
```

Результат достаточно очевиден.

pfile

Функция `pfile` передает файл, содержащий указанный объект, программе постраничного просмотра, если этот файл будет найден:

```
In [1]: import os

In [2]: pfile os
```

```
r"""OS routines for Mac, NT, or Posix depending on what system we're on.

This exports:
  - all functions from posix, nt, os2, mac, or ce, e.g. unlink, stat, etc.

<... дальнейшие результаты обрезаны ...>
```

В этом примере открывается файл модуля `os` и передается программе `less`. Это может оказаться удобным, если вы пытаетесь понять, почему тот или иной фрагмент программного кода ведет себя тем или иным способом. Эта функция не работает, если доступным файлом является пакет или файл с байт-кодом *.pyc*.

Ту же информацию, что выводят специальные функции `%pdef`, `%pdoc` и `%pfile`, можно получить с помощью оператора `??`. Причем использование оператора `??` предпочтительнее.

pinfo

Функция `pinfo` и родственные ей утилиты настолько удобны, что сложно представить, как можно обходиться без них. Функция `pinfo` предоставляет такую информацию, как тип, базовый класс, пространство имен и строка документирования. Если представить, что у нас имеется модуль, содержащий следующее:

```
#!/usr/bin/env python

class Foo:
    """мой класс Foo"""
    def __init__(self):
        pass

class Bar:
    """мой класс Bar"""
    def __init__(self):
        pass

class Bam:
    """мой класс Bam"""
    def __init__(self):
        pass
```

то можно было бы запросить информацию непосредственно о модуле:

```
In [1]: import some_module

In [2]: pinfo some_module
Type:           module
Base Class:     <type 'module'>
String Form:    <module 'some_module' from 'some_module.py'>
Namespace:      Interactive
File:           /home/jmjones/code/some_module.py
Docstring:
    <no docstring>
```

Об определенном классе в этом модуле:

```
In [3]: pinfo some_module.Foo
Type:           classobj
String Form:    some_module.Foo
Namespace:      Interactive
File:           /home/jmjones/code/some_module.py
Docstring:
    мой класс Foo

Constructor information:
Definition:    some_module.Foo(self)
```

Об экземпляре одного из классов:

```
In [4]: f = some_module.Foo()

In [5]: pinfo f
Type:          instance
Base Class:    some_module.Foo
String Form:   <some_module.Foo instance at 0x86e9e0>
Namespace:     Interactive
Docstring:
    мой класс Foo
```

Оператор `?`, стоящий перед или после имени объекта, позволит получить ту же информацию, которую выводит функция `pinfo`:

```
In [6]: ? f
Type:           instance
Base Class:     some_module.Foo
String Form:    <some_module.Foo instance at 0x86e9e0>
Namespace:      Interactive
Docstring:
    мой класс Foo

In [7]: f ?
Type:           instance
Base Class:     some_module.Foo
String Form:    <some_module.Foo instance at 0x86e9e0>
Namespace:      Interactive
Docstring:
    мой класс Foo
```

А два знака вопроса (`??`), помещенные перед или после имени объекта, позволят получить еще больше информации:

```
In [8]: some_module.Foo ??
Type:           classobj
String Form:    some_module.Foo
Namespace:      Interactive
File:           /home/jmjones/code/some_module.py
Source:
class Foo:
    """мой класс Foo"""
```

```
        def __init__(self):
            pass
Constructor information:
Definition: some_module.Foo(self)
```

Оператор ?? выводит ту же информацию, которую дает функция pinfo, плюс исходный программный код реализации запрошенного объекта. Поскольку в этом примере мы запросили информацию об определенном классе, оператор ?? вывел исходный программный код только для этого объекта, а не весь файл целиком. Эта особенность оболочки IPython используется значительно чаще, чем любая другая ее особенность.

psource

Функция psource выводит исходный программный код указанного объекта, будь то модуль или элемент модуля, такой как класс или функция. Для отображения исходного программного кода используется программа постраничного просмотра. Ниже приводится пример использования psource для просмотра содержимого модуля:

```
In [1]: import some_other_module

In [2]: psource some_other_module
#!/usr/bin/env python

class Foo:
    """мой класс Foo"""
    def __init__(self):
        pass

class Bar:
    """мой класс Bar"""
    def __init__(self):
        pass

class Bam:
    """мой класс Bam"""
    def __init__(self):
        pass

def baz():
    """моя функция baz"""
    return None
```

Ниже приводится пример использования функции psource для просмотра исходного кода класса в модуле:

```
In [3]: psource some_other_module.Foo
class Foo:
    """мой класс Foo"""
    def __init__(self):
        pass
```

и в следующем примере функция psource используется для вывода исходного кода функции:

```
In [4]: psource some_other_module.baz
def baz():
    """моя функция baz"""
    return None
```

psearch

Специальная функция psearch позволяет отыскать объект на языке Python по имени, с возможностью использования шаблонных символов. Здесь мы лишь коротко опишем функцию psearch. Если вам потребуется дополнительная информация о ней, вы можете обратиться к документации по специальным функциям, введя команду magic в строке приглашения IPython и отыскав описание функции psearch в списке, отсортированном по алфавиту.

Для начала объявим следующие объекты:

```
In [1]: a = 1

In [2]: aa = "one"

In [3]: b = 2

In [4]: bb = "two"

In [5]: c = 3

In [6]: cc = "three"
```

Мы можем отыскать все объекты, имена которых начинаются с символа a, b или c, следующим способом:

```
In [7]: psearch a*
a
aa
abs
all
any
apply

In [8]: psearch b*
b
basestring
bb
bool
buffer

In [9]: psearch c*
c
callable
cc
chr
```

```
classmethod
cmp
coerce
compile
complex
copyright
credits
```

Обратите внимание, что помимо наших объектов a, aa, b, bb, c и cc были найдены еще и встроенные объекты.

Оператор ? может рассматриваться как приблизительный эквивалент функции psearch. Например:

```
In [2]: import os

In [3]: psearch os.li*
os.linesep
os.link
os.listdir

In [4]: os.li*?
os.linesep
os.link
os.listdir
```

То есть вместо psearch мы вполне можем использовать *?.

Функция psearch имеет дополнительные параметры: -s позволяет включить, а -e – исключить из поиска указанное пространство из относящихся к этой функции пространств имен. В пространства имен входят builtin, user, user_global, internal и alias. По умолчанию функция psearch производит поиск в пространствах имен builtin и user. Чтобы произвести поиск только в пространстве имен user, можно было бы передать функции psearch параметр -e builtin, который исключит из поиска пространство имен builtin. Использование этих параметров несколько неочевидно, но имеет некоторый смысл. По умолчанию функция psearch производит поиск в пространствах имен builtin и user, поэтому, если использовать параметр -s user, поиск по-прежнему будет производиться в пространствах имен builtin и user. В следующем примере мы еще раз выполнили поиск, но на этот раз исключили пространство встроенных имен builtin:

```
In [10]: psearch -e builtin a*
a
aa

In [11]: psearch -e builtin b*
b
bb

In [12]: psearch -e builtin c*
c
cc
```

Кроме того, функция psearch позволяет отыскивать объекты указанных типов. Ниже мы выполнили поиск целочисленных объектов в пространстве имен user:

```
In [13]: psearch -e builtin * int
a
b
c
```

В следующем примере произведен поиск строк:

```
In [14]: psearch -e builtin * string
__
___
__name__
aa
bb
cc
```

Объекты __ и ___, которые были найдены здесь, являются сокращениями IPython. Объект __name__ – это специальная переменная, которая хранит имя модуля. Если переменная __name__ содержит строку '__main__', это означает, что модуль выполняется как самостоятельный сценарий, а не импортируется другим модулем.

who

Оболочка IPython предоставляет множество способов получения списков интерактивных объектов. Первый из них – функция who. Ниже приводится продолжение предыдущего примера с переменными a, aa, b, bb, c и cc и использованием функции who:

```
In [15]: who
a       aa      b       bb      c       cc
```

Эта функция не содержит никаких подвохов, она просто выводит перечень всех объектов, которые были определены в интерактивном режиме. Функцию who можно использовать для выборки переменных определенных типов, например:

```
In [16]: who int
a       b       c

In [17]: who str
aa      bb      cc
```

who_ls

Функция who_ls похожа на функцию who, за исключением того, что она не выводит перечень имен соответствующих переменных, а возвращает список. Ниже приводится пример использования функции who_ls без аргументов:

```
In [18]: who_ls
Out[18]: ['a', 'aa', 'b', 'bb', 'c', 'cc']
```

А в следующем примере производится выборка объектов определенного типа:

```
In [19]: who_ls int
Out[19]: ['a', 'b', 'c']
In [20]: who_ls str
Out[20]: ['aa', 'bb', 'cc']
```

Функция `who_ls` возвращает список имен, поэтому вы можете получить доступ к нему с помощью переменной `_`, которая содержит «последний выведенный результат». Ниже демонстрируется способ обхода последнего полученного списка с соответствующими именами переменных:

```
In [21]: for n in _:
   ....:     print n
   ....:
   ....:
aa
bb
cc
```

whos

Функция `whos` похожа на функцию `who`, но в отличие от последней функция `whos` выводит информацию в табличном виде. Ниже приводится пример использования функции `whos` без аргументов:

```
In [22]: whos
Variable   Type    Data/Info
----------------------------
a          int     1
aa         str     one
b          int     2
bb         str     two
c          int     3
cc         str     three
n          str     cc
```

Так же, как и функция `who`, она способна отбирать переменные в соответствии с указанным типом:

```
In [23]: whos int
Variable   Type    Data/Info
----------------------------
a          int     1
b          int     2
c          int     3
```

```
In [24]: whos str
Variable   Type    Data/Info
---------------------------
aa         str     one
bb         str     two
cc         str     three
n          str     cc
```

История

В оболочке IPython существует два способа получения доступа к истории вводившихся команд. Первый основан на использовании поддержки библиотеки readline, а второй – на использовании специальной функции `hist`.

Поддержка readline

В оболочке IPython обеспечивается доступ ко всем функциональным возможностям, которые может предоставить приложение, обладающее поддержкой библиотеки readline. Если вы привыкли пользоваться управляющими комбинациями для выполнения поиска по истории команд в оболочке Bash, значит у вас не будет проблем с использованием той же самой функциональности в IPython. В примере ниже определяется несколько переменных, а затем производится поиск по истории команд:

```
In [1]: foo = 1

In [2]: bar = 2

In [3]: bam = 3

In [4]: d = dict(foo=foo, bar=bar, bam=bam)

In [5]: dict2 = dict(d=d, foo=foo)

In [6]: <CTRL-r>

(reverse-i-search)`fo': dict2 = dict(d=d, foo=foo)

<CTRL-r>

(reverse-i-search)`fo': d = dict(foo=foo, bar=bar, bam=bam)
```

Здесь мы нажали комбинацию клавиш Ctrl-r, затем ввели строку `fo`, которая выступает в качестве критерия поиска. В результате была получена строка, которую мы ввели в приглашении IPython `In[5]`. Пользуясь поддержкой поиска в библиотеке readline, мы вновь нажали комбинацию Ctrl-r и получили строку, которую мы ввели в приглашении IPython `In[4]`.

Есть еще некоторые комбинации клавиш, которые можно использовать при наличии поддержки readline, но мы коснемся их очень кратко. Комбинация Ctrl-a переносит курсор в начало строки, комбинация

Ctrl-e – в конец строки. Комбинация Ctrl-f вызывает перемещение курсора на один символ вперед (forward), а комбинация Ctrl-b – на один символ назад (backward). Комбинация Ctrl-d удаляет (delete) один символ под курсором, а комбинация Ctrl-h – удаляет один символ левее курсора (аналогично действию клавиши Backspace (забой)). Комбинация Ctrl-p вызывает перемещение на одну команду назад, к началу истории, а комбинация Ctrl-n – на одну команду вперед, к концу истории. Дополнительную информацию о возможностях readline можно получить в справочном руководстве *nix-систем с помощью команды `man readline`.

Функция hist

В дополнение к возможности доступа к истории команд посредством библиотеки readline оболочка IPython также предоставляет для этого свою собственную специальную функцию с именем `history` и сокращенный вариант имени `hist`. При вызове без параметров функция `hist` выводит последовательный список команд, вводившихся пользователем. По умолчанию строки в этом списке пронумерованы. В следующем примере мы определили несколько переменных, перешли в другой каталог и затем вызвали функцию `hist`:

```
In [1]: foo = 1

In [2]: bar = 2

In [3]: bam = 3

In [4]: cd /tmp
/tmp

In [5]: hist
1: foo = 1
2: bar = 2
3: bam = 3
4: _ip.magic("cd /tmp")
5: _ip.magic("hist ")
```

Четвертый и пятый элементы в списке выше – это вызовы специальных функций. Обратите внимание на то, как они были изменены оболочкой IPython, благодаря чему можно видеть, что в действительности команды выполняются через вызов оболочкой функции `magic()`.

Чтобы подавить вывод номеров строк, можно использовать параметр `-n`. Ниже приводится пример использования функции `hist` с параметром `-n`:

```
In [6]: hist -n
foo = 1
bar = 2
bam = 3
_ip.magic("cd /tmp")
```

```
_ip.magic("hist ")
_ip.magic("hist -n")
```

Это очень удобно, когда при работе в IPython возникает необходимость скопировать блок программного кода из оболочки IPython и вставить его в текстовый редактор.

Параметр `-t` возвращает «преобразованное» (translated) представление истории, где показано, как в действительности выглядят команды, введенные в оболочке IPython. Этот параметр установлен по умолчанию. Ниже приводится история команд, которая образовалась к настоящему моменту, полученная с параметром `-t`:

```
In [7]: hist -t
1: foo = 1
2: bar = 2
3: bam = 3
4: _ip.magic("cd /tmp")
5: _ip.magic("hist ")
6: _ip.magic("hist -n")
7: _ip.magic("hist -t")
```

При использовании параметра `-r` выводится история команд в «непосредственном (сыром) виде» (raw) и отображаются команды в том виде, в каком они вводились. Ниже приводится результат применения параметра `-r`:

```
In [8]: hist -r
1: foo = 1
2: bar = 2
3: bam = 3
4: cd /tmp
5: hist
6: hist -n
7: hist -t
8: hist -r
```

Параметр `-g` функции обеспечивает возможность поиска в истории по заданному шаблону. Ниже приводится пример использования параметра `-g`, чтобы отыскать в истории все вхождения слова `hist`:

```
In [9]: hist -g hist
0187: hist
0188: hist -n
0189: hist -g import
0190: hist -h
0191: hist -t
0192: hist -r
0193: hist -d
0213: hist -g foo
0219: hist -g hist
===
^shadow history ends, fetch by %rep <number> (must start with 0)
```

```
=== start of normal history ===
5 : _ip.magic("hist ")
6 : _ip.magic("hist -n")
7 : _ip.magic("hist -t")
8 : _ip.magic("hist -r")
9 : _ip.magic("hist -g hist")
```

Обратите внимание на слова «shadow history» (теневая история), появившиеся в предыдущем примере. «Теневая история» – это история всех команд, которые вводились когда-либо. Строки с элементами этой истории отображаются в самом начале списка и начинаются с нуля. Строки с элементами истории текущего сеанса отображаются в конце списка и не начинаются с нуля.

История результатов

И в стандартной оболочке Python, и в оболочке IPython имеется возможность доступа не только к истории вводившихся команд, но к истории результатов. Первый способ доступа заключается в использовании специальной переменной с именем _, которая содержит «последний выведенный результат». Ниже приводится пример использования переменной _ в IPython:

```
In [1]: foo = "foo_string"

In [2]: _

Out[2]: ''

In [3]: foo

Out[3]: 'foo_string'

In [4]: _

Out[4]: 'foo_string'

In [5]: a = _

In [6]: a

Out[6]: 'foo_string'
```

Когда в приглашении `In [1]` мы определили переменную `foo`, переменная _ в `In [2]` вернула пустую строку. Когда мы вывели значение переменной `foo` в `In [3]`, переменная _ в `In [4]` вернула полученный выше результат. А операция в `In [5]` показала, что имеется возможность сохранять результат в переменной.

Ниже приводится тот же самый пример, выполненный в стандартной оболочке Python:

```
>>> foo = "foo_string"
>>> _
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
```

```
NameError: name '_' is not defined
>>> foo
'foo_string'
>>> _
'foo_string'
>>> a = _
>>> a
'foo_string'
```

Здесь, в стандартной оболочке Python, наблюдаем практически ту же самую картину, что и в оболочке IPython, за исключением того, что при попытке обратиться к имени _ до того, как что-нибудь будет выведено, возбуждается исключение `NameError`.

Оболочка IPython поднимает концепцию «последнего выведенного результата» на новый уровень. В разделе «Выполнение системных команд» было дано описание операторов `!` и `!!` и говорилось, что результат работы оператора `!!` нельзя сохранить в переменной, но позднее его можно использовать. Проще говоря, у вас имеется доступ к любым результатам с помощью символа подчеркивания (_), вслед за которым следует число в соответствии с синтаксисом `_[0-9]*`. Число должно соответствовать результату в строке `Out[0-9]*`.

Чтобы продемонстрировать, как действует этот прием, мы сначала выведем списки файлов, но при этом ничего не будем делать с полученными результатами:

```
In [1]: !!ls apa*py

Out[1]: SList (.p, .n, .l, .s, .grep(), .fields() available). Value:
0: apache_conf_docroot_replace.py
1: apache_log_parser_regex.py
2: apache_log_parser_split.py

In [2]: !!ls e*py

Out[2]: SList (.p, .n, .l, .s, .grep(), .fields() available). Value:
0: elementtree_system_profile.py
1: elementtree_tomcat_users.py

In [3]: !!ls t*py

Out[3]: SList (.p, .n, .l, .s, .grep(), .fields() available). Value:
0: test_apache_log_parser_regex.py
1: test_apache_log_parser_split.py
```

Теперь у нас должна иметься возможность доступа к `Out[1-3]` с помощью `_1`, `_2` и `_3`. Чтобы было более понятно, мы присвоим эти значения переменным с говорящими именами:

```
In [4]: apache_list = _1

In [5]: element_tree_list = _2

In [6]: tests = _3
```

Теперь `apache_list`, `tree_list` и `tests` содержат те же элементы, которые были выведены в строках `Out[1]`, `Out[2]` и `Out[3]`, соответственно:

```
In [7]: apache_list

Out[7]: SList (.p, .n, .l, .s, .grep(), .fields() available). Value:
0: apache_conf_docroot_replace.py
1: apache_log_parser_regex.py
2: apache_log_parser_split.py

In [8]: element_tree_list

Out[8]: SList (.p, .n, .l, .s, .grep(), .fields() available). Value:
0: elementtree_system_profile.py
1: elementtree_tomcat_users.py

In [9]: tests

Out[9]: SList (.p, .n, .l, .s, .grep(), .fields() available). Value:
0: test_apache_log_parser_regex.py
1: test_apache_log_parser_split.py
```

Обобщим сказанное: в оболочке IPython имеется возможность обращаться к выведенным ранее результатам либо через специальную переменную _, либо через _ с явно указанным номером полученного ранее результата.

Автоматизация и сокращения

Оболочка IPython делает достаточно много, чтобы повысить производительность труда, и кроме этого она предоставляет ряд функций и особенностей, помогающих автоматизировать решение задач в IPython.

alias

Для начала упомянем специальную функцию `alias`. Мы уже рассматривали ее выше в этой главе, поэтому не будем повторно описывать принципы ее использования. Но нам хотелось бы напомнить, что функция `alias` способна не только помочь использовать системные команды системы *nix в оболочке IPython, но также может оказать помощь в автоматизации решения задач.

macro

Функция macro позволяет определять блоки программного кода, которые могут выполняться позднее, в составе любого программного кода, с которым вам придется работать. Макроопределения, создаваемые с помощью специальной функции `macro`, выполняются в текущем контексте вашего программного кода. Если у вас имеется некоторая последовательность инструкций, которую вы часто используете для обработки своих файлов, вы можете создать макроопределение, которое

будет выполнять эту работу. Чтобы получить представление о том, как с помощью макроопределения можно выполнять обработку списка файлов, рассмотрим следующий пример:

```
In [1]: dirlist = []

In [2]: for f in dirlist:
   ...:     print "working on", f
   ...:     print "done with", f
   ...:     print "moving %s to %s.done" % (f, f)
   ...:     print "*" * 40
   ...:
   ...:

In [3]: macro procdir 2
Macro `procdir` created. To execute, type its name (without quotes).
Macro contents:
for f in dirlist:
    print "working on", f
    print "done with", f
    print "moving %s to %s.done" % (f, f)
    print "*" * 40
```

К моменту создания цикла в `In [2]` в `dirlist` не было ни одного элемента, чтобы их можно было обойти в цикле, но так как мы предполагаем, что позднее в `dirlist` появятся элементы, мы создали макрокоманду с именем `procdir`, которая выполняет обход списка в цикле. Макрокоманда создается в соответствии с синтаксисом: `macro macro_name range_of_lines`, где под `range_of_lines` подразумевается список строк истории команд, которые должны быть добавлены в макроопределение. Строки в этом списке должны определяться номерами или диапазонами номеров (например, `1-4`) строк и отделяться друг от друга пробелами.

В следующем примере мы создали список имен файлов и сохранили его в переменной `dirlist`, а затем выполнили макрокоманду `procdir`. Макрокоманда выполнит обход списка файлов в `dirlist`:

```
In [4]: dirlist = ['a.txt', 'b.txt', 'c.txt']

In [5]: procdir
------> procdir()
working on a.txt
done with a.txt
moving a.txt to a.txt.done
****************************************
working on b.txt
done with b.txt
moving b.txt to b.txt.done
****************************************
working on c.txt
done with c.txt
moving c.txt to c.txt.done
****************************************
```

После того как макрокоманда будет определена, ее можно будет отредактировать с помощью функции `edit`. В результате этого будет открыт текстовый редактор. Очень удобно иметь возможность выполнить отладку макрокоманды, добиваясь правильной ее работы, прежде чем сохранить ее.

store

Вы можете сохранить свои макрокоманды и простые переменные с помощью специальной функции `store`. Она имеет следующий стандартный формат: `store variable`. Кроме того, функция `store` может принимать дополнительные параметры, которые могут оказаться для вас полезными: вызов `store -d variable` удалит указанную переменную из списка сохраняемых; параметр `-z` удалит все сохраняемые переменные; а параметр `-r` выполнит повторную загрузку всех сохраняемых переменных.

reset

Функция `reset` удаляет все переменные из интерактивного пространства имен. В следующем примере определяются три переменные, затем вызывается функция `whos`, чтобы убедиться в их присутствии, затем выполняется очистка пространства имен с помощью функции `reset` и повторно вызывается функция `whos`, чтобы убедиться в том, что переменные исчезли:

```
In [1]: a = 1

In [2]: b = 2

In [3]: c = 3

In [4]: whos
Variable   Type    Data/Info
---------------------------
a          int     1
b          int     2
c          int     3

In [5]: reset
Once deleted, variables cannot be recovered. Proceed (y/[n])? y
(После удаления переменные нельзя будет восстановить. Продолжить (y/[n])?)

In [6]: whos
Interactive namespace is empty.
(Интерактивное пространство имен очищено.)
```

run

Функция `run` выполняет указанный файл в оболочке IPython. Это, кроме всего, позволяет работать с модулями на языке Python во внешнем текстовом редакторе и интерактивно тестировать внесенные изменения

в IPython. После выполнения указанной программы управление возвращается в оболочку IPython. Функция `run` имеет следующий формат записи: `run options specified_file args`.

При использовании параметра `-n` переменная `__name__` модуля получает в качестве значения название модуля, а не строку `'__main__'`. Это приводит к тому, что модуль выполняется так, как если бы он был просто импортирован.

При использовании параметра `-i` модуль выполняется в текущем пространстве имен оболочки IPython, благодаря чему модуль получает доступ ко всем переменным, которые были определены.

При использовании параметра `-e` оболочка IPython будет игнорировать вызов функции `sys.exit()` и исключение `SystemExit`. Даже если они будут иметь место, оболочка IPython продолжит свою работу.

При использовании параметра `-t` оболочка IPython выведет информацию о времени выполнения модуля.

При использовании параметра `-d` указанный модуль будет запущен под управлением отладчика Python (pdb).

При использовании параметра `-p` указанный модуль будет запущен под управлением профилировщика.

save

Функция `save` сохраняет указанные строки ввода в указанный файл. Порядок использования функции `save`: `save options filename lines`. Строки могут указываться в том же формате, что и в функции `macro`. Единственный дополнительный параметр `-r` определяет, что в файл следует сохранить строки в непреобразованном виде, то есть том виде, в каком они вводились. По умолчанию строки сохраняются в преобразованном, стандартном для языка Python, виде.

rep

Последняя функция, используемая для автоматизации решения задач, – это функция `rep`. Функция `rep` может принимать ряд параметров, которые вы найдете полезными. Вызов функции `rep` без параметров возвращает последний вычисленный результат и помещает строковое его представление в следующей строке ввода. Например:

```
In [1]: def format_str(s):
   ...:     return "str(%s)" % s
   ...:

In [2]: format_str(1)

Out[2]: 'str(1)'

In [3]: rep

In [4]: str(1)
```

Вызов функции `rep` в строке `In [3]` привел к вставке текста в строке `In [4]`. Такая ее особенность позволяет программно генерировать ввод в оболочке IPython. Это особенно удобно при использовании комбинаций макрокоманд и генераторов.

Обычно функция `rep` без параметров используется при редактировании без использования мыши. Если у вас имеется переменная, содержащая некоторое значение, с помощью этой функции его можно будет редактировать непосредственно. В качестве примера представим, что у нас имеется функция, которая возвращает каталог *bin*, куда был установлен некоторый пакет. Мы сохраняем каталог *bin* в переменной с именем `a`:

```
In [2]: a = some_blackbox_function('squiggly')

In [3]: a

Out[3]: '/opt/local/squiggly/bin'
```

Если вызвать функцию `rep` прямо сейчас, мы получим строку `/opt/local/squiggly/bin` в новой строке ввода, с мигающим курсором в конце строки, приглашающим нас к ее редактированию:

```
In [4]: rep

In [5]: /opt/local/squiggly/bin<мигающий курсор>
```

Если нам требуется сохранить не каталог *bin*, а корневой каталог пакета, мы можем просто удалить `bin` в конце строки, окружить строку кавычками и добавить в начало строки ввода имя новой переменной и оператор присваивания:

```
In [5]: new_a = '/opt/local/squiggly'
```

Теперь у нас имеется новая переменная, содержащая строку с именем корневого каталога данного пакета.

Несомненно, мы могли бы просто скопировать и вставить эту строку, но для этого пришлось бы выполнить больший объем работы. Зачем нам отвлекаться от столь удобной клавиатуры, чтобы дотянуться до мыши? Теперь вы можете использовать переменную `new_a` в качестве корневого каталога для выполнения любых необходимых действий с пакетом.

Когда функции `rep` в качестве параметра передается число, она выбирает значение соответствующей строки ввода из истории команд, вставляет ее в следующую строку ввода и помещает курсор в конец этой строки. Это бывает удобно для запуска, редактирования и повторного запуска отдельных строк или даже небольших блоков программы. Например:

```
In [1]: map = (('a', '1'), ('b', '2'), ('c', '3'))

In [2]: for alph, num in map:
```

```
   ...:     print alph, num
   ...:
   ...:
a 1
b 2
c 3
```

Теперь нам нужно отредактировать строку ввода `In [2]` так, чтобы выводились значения, умноженные на 2. Для этого можно снова ввести цикл `for` или воспользоваться функцией `rep`:

```
In [3]: rep 2

In [4]: for alph, num in map:
    print alph, int(num) * 2
   ...:
   ...:
a 2
b 4
c 6
```

Кроме того, функция `rep` способна принимать диапазоны строк. Синтаксис диапазона аналогичен тому, что используется в функции `macro`, которая уже рассматривалась выше в этой главе. Когда функции `rep` передается диапазон строк, они выполняются немедленно, например:

```
In [1]: i = 1

In [2]: i += 1

In [3]: print i
2

In [4]: rep 2-3
lines [u'i += 1\nprint i\n']
3

In [7]: rep 2-3
lines [u'i += 1\nprint i\n']
4
```

Здесь в строках с `In [1]` по `In [3]` мы определили переменную, увеличили ее на 1, и вывели текущее значение. В строках `In [4]` и `In [7]` мы предложили функции `rep` повторить строки 2 и 3. Обратите внимание на отсутствие двух строк (5 и 6) – эти строки были выполнены после строки `In [4]`.

Последний параметр функции `rep`, который мы рассмотрим, – строка. Этот случай можно выразить словами: «передача слова функции `rep`» или даже: «передача функции `rep` искомой строки без кавычек». Например:

```
In [1]: a = 1

In [2]: b = 2
```

```
In [3]: c = 3

In [4]: rep a

In [5]: a = 1
```

Здесь мы определили несколько переменных и потребовали от функции `rep` повторить строку, в которой присутствует слово «a». В результате мы получили строку, введенную в приглашении `In [1]`, и получили возможность отредактировать ее и запустить повторно.

В заключение

Оболочка IPython является самым ценным инструментом в нашем арсенале. Мастерство владения ее возможностями напоминает мастерство владения текстовым редактором: чем большим опытом вы обладаете, тем быстрее будете решать утомительные задачи. Даже несколько лет тому назад, когда мы только начали использовать оболочку IPython, она уже была весьма мощным инструментом. С тех пор ее мощь увеличилась еще больше. Функция `grep` и обработка строк – это первые две особенности, которые сразу же приходят на ум, когда мы задумываемся о по-настоящему полезных и мощных возможностях, перечень которых непрерывно продолжает пополняться усилиями сообщества IPython. Мы настоятельно рекомендуем поближе познакомиться с оболочкой IPython. Освоение ее – это надежные инвестиции в будущее, о которых вам не придется сожалеть.

3

Текст

Практически каждому системному администратору приходится иметь дело с текстом в той или иной форме, например, с файлами журналов, данными приложений, с XML-, HTML- и конфигурационными файлами или с выводом некоторых команд. Обычно для работы вполне хватает таких утилит, как `grep` и `awk`, но иногда для решения сложных задач необходим более элегантный и выразительный инструмент. Когда возникает потребность создать файл с данными, извлеченными из других файлов, часто бывает достаточно перенаправить вывод процесса обработки (здесь опять приходят на ум `grep` и `awk`) в файл. Но иногда складываются ситуации, когда для выполнения задания требуется инструмент с более широкими возможностями.

Как мы уже говорили во «Введении», наш опыт показывает, что язык Python можно рассматривать как более элегантный, выразительный и расширяемый, чем Perl, Bash или другие языки программирования, которые мы использовали в своей практике. Подробное описание причин, почему мы оцениваем Python более высоко, чем Perl или Bash (то же самое относится к `sed` и `awk`), приводится в главе 1. Стандартная библиотека языка Python, особенности языка и встроенные типы представляют собой мощные средства чтения текстовых файлов, манипулирования текстом и извлечения информации из текстовых файлов. Язык Python и стандартная библиотека обладают богатыми и гибкими функциональными возможностями обработки текста с помощью строкового типа, файлового типа и модуля регулярных выражений. Недавнее пополнение стандартной библиотеки – модуль ElementTree – чрезвычайно удобен при работе с данными в формате XML. В этой главе мы покажем, как эффективно использовать стандартную библиотеку и встроенные компоненты при работе с текстовой информацией.

Встроенные компоненты Python и модули

str

Строка – это просто последовательность символов. При любой работе с текстовой информацией вы почти наверняка вынуждены будете работать с ней как со строковым объектом или с последовательностью строковых объектов. Строковый тип, `str`, – это мощное, гибкое средство манипулирования строковыми данными. В этом разделе показывается, как создавать строки и какие операции можно выполнять над ними после их создания.

Создание строк

Обычно строка создается путем заключения некоторого текста в кавычки:

```
In [1]: string1 = 'This is a string'

In [2]: string2 = "This is another string"

In [3]: string3 = '''This is still another string'''

In [4]: string4 = """And one more string"""

In [5]: type(string1), type(string2), type(string3), type(string4)
Out[5]: (<type 'str'>, <type 'str'>, <type 'str'>, <type 'str'>)
```

Апострофы и кавычки, обычные и тройные, обозначают одно и то же: все они создают объект типа `str`. Апострофы и кавычки идентичны по своему действию и являются взаимозаменяемыми. Этим язык Python отличается от командных оболочек UNIX, где апострофы и кавычки не являются взаимозаменяемыми. Например:

```
jmjones@dink:~$ FOO=sometext
jmjones@dink:~$ echo "Here is $FOO"
Here is sometext
jmjones@dink:~$ echo 'Here is $FOO'
Here is $FOO
```

В языке Perl апострофы и кавычки также не могут замещать друг друга при создании строк. Ниже приводится похожий пример на языке Perl.

```
#!/usr/bin/perl

$FOO = "some_text";
print "-- $FOO --\n";
print '-- $FOO --\n';
```

И вот какие результаты дает этот небольшой сценарий на языке Perl:

```
jmjones@dinkgutsy:code$ ./quotes.pl
-- some_text --
-- $FOO --\njmjones@dinkgutsy:code$
```

Это различие отсутствует в языке Python. Право определять различия Python оставляет за программистом. Например, вы можете использовать апострофы, когда внутри строки должны находиться кавычки и вам не хотелось бы экранировать их (символом обратного слеша). Точно так же вы можете использовать кавычки, когда внутри строки должны присутствовать апострофы и вам не хотелось бы экранировать их, как показано в примере 3.1.

Пример 3.1. Кавычки и апострофы в языке Python

```
In [1]: s = "This is a string with 'quotes' in it"

In [2]: s
Out[2]: "This is a string with 'quotes' in it"

In [3]: s = 'This is a string with \'quotes\' in it'

In [4]: s
Out[4]: "This is a string with 'quotes' in it"

In [5]: s = 'This is a string with "quotes" in it'

In [6]: s
Out[6]: 'This is a string with "quotes" in it'

In [7]: s = "This is a string with \"quotes\" in it"

In [8]: s
Out[8]: 'This is a string with "quotes" in it'
```

Обратите внимание, что во 2-й и 4-й строках вывода (`Out [4]` и `Out [8]`) включение в строку экранированных кавычек того же типа, что и окружающие строку, привело при выводе к изменению типа наружных кавычек. (В действительности это приводит отображение строки к «правильному» применению кавычек разных типов.)

Иногда бывает необходимо, чтобы в одной строке объединялось несколько строк. Иногда эту проблему можно решить, вставляя символ `\n` там, где необходимо создать разрыв строки, но это довольно неудобный способ. Другая, более ясная альтернатива заключается в использовании тройных кавычек, которые позволяют определять многострочный текст. В примере 3.2 демонстрируется неудачная попытка использовать апострофы для определения многострочного текста и успешная попытка использовать тройные апострофы.

Пример 3.2. Тройные кавычки

```
In [6]: s = 'this is
------------------------------------------------------------
  File "<ipython console>", line 1
    s = 'this is
                ^
SyntaxError: EOL while scanning single-quoted string
(SyntaxError: обнаружен конец строки при интерпретации строки в апострофах)
```

```
In [7]: s = '''this is a
   ...: multiline string'''

In [8]: s
Out[8]: 'this is a\nmultiline string'
```

Помимо этого существует еще один способ обозначения строк, которые в языке Python называются «сырыми» строками. Сырые строки создаются добавлением символа `r` непосредственно перед открывающей кавычкой. По сути дела, сырые строки отличаются от обычных строк тем, что в сырых строках Python не интерпретирует экранированные последовательности символов, тогда как в обычных строках они интерпретируются. При интерпретации экранированных последовательностей в языке Python соблюдается практически тот же набор правил, который описывается стандартом языка C. Например, в обычных строках последовательность `\t` интерпретируется как символ табуляции, `\n` – как символ новой строки и `\r` – как перевод строки. В табл. 3.1 приводится список экранированных последовательностей в языке Python.

Таблица 3.1. Экранированные последовательности в языке Python

Последовательность	Интерпретируется как
`\newline`	Игнорируется
`\\`	Символ обратного слеша
`\'`	Апостроф
`\"`	Кавычка
`\a`	ASCII-символ звукового сигнала
`\b`	ASCII-символ забоя
`\f`	ASCII-символ перевода формата (страницы)
`\n`	ASCII-символ новой строки
`\N{имя}`	Именованный символ Юникода (только для строк в кодировке Юникод)
`\r`	ASCII-символ возврата каретки
`\t`	ASCII-символ горизонтальной табуляции
`\uxxxx`	Шестнадцатеричный код 16-битового символа (только для строк в кодировке Юникод)
`\Uxxxxxxxx`	Шестнадцатеричный код 32-битового символа (только для строк в кодировке Юникод)
`\v`	ASCII-символ вертикальной табуляции
`\ooo`	Восьмеричный код символа
`\xhh`	Шестнадцатеричный код символа

Об экранированных последовательностях и сырых строках стоит помнить, особенно, когда приходится иметь дело с регулярными выражениями, к которым мы подойдем далее в этой главе. В примере 3.3 демонстрируется использование экранированных последовательностей и неформатированных строк.

Пример 3.3. Экранированные последовательности и сырые строки

```
In [1]: s = '\t'

In [2]: s
Out[2]: '\t'

In [3]: print s


In [4]: s = r'\t'

In [5]: s
Out[5]: '\\t'

In [6]: print s
\t

In [7]: s = '''\t'''

In [8]: s
Out[8]: '\t'

In [9]: print s

In [10]: s = r'''\t'''

In [11]: s
Out[11]: '\\t'

In [12]: print s
\t

In [13]: s = r'\''

In [14]: s
Out[14]: "\\'"

In [15]: print s
\'
```

Когда выполняется интерпретация экранированных последовательностей, `\t` превращается в символ табуляции. Когда интерпретация не выполняется, экранированная последовательность `\t` воспринимается, как строка из двух символов, `\` и `t`. Строки, окруженные кавычками или апострофами, обычными или тройными, подразумевают, что последовательность `\t` будет интерпретироваться как символ табуляции. Если те же самые строки предваряются символом `r`, последовательность `\t` интерпретируется как два символа, `\` и `t`.

Еще один фокус этого примера – различия между `__repr__` и `__str__`. Когда имя переменной вводится в строке приглашения оболочки IPython

и нажимается клавиша Enter, значение переменной отображается вызовом метода __repr__. Когда вводится инструкция print, которой передается имя переменной, и нажимается клавиша Enter, переменная отображается вызовом метода __str__. Инструкция print интерпретирует экранированные последовательности в строке и отображает их соответствующим образом. Подробнее о __repr__ и __str__ рассказывается в главе 2, в разделе «Базовые понятия».

Встроенные методы извлечения строковых данных

Строки в языке Python – это объекты, поэтому они имеют методы, которые могут вызываться для выполнения определенных операций. Однако под «методами» мы подразумеваем не только методы, которыми обладает тип str, но и любые другие способы, позволяющие извлекать данные из объектов типа str. Сюда входят все методы типа str, а также операторы in и not in, приведенные в примере, следующем ниже.

С технической точки зрения, операторы проверки условия in и not in вызывают метод __contains__() объекта str. За дополнительной информацией о том, как работают эти операторы, обращайтесь к приложению. Операторы in и not in могут использоваться для проверки, является ли некоторая строка частью другой строки, как показано в примере 3.4.

Пример 3.4. Операторы in и not in

```
In [1]: import subprocess

In [2]: res = subprocess.Popen(['uname', '-sv'], stdout=subprocess.PIPE)

In [3]: uname = res.stdout.read().strip()

In [4]: uname
Out[4]: 'Linux #1 SMP Tue Feb 12 02:46:46 UTC 2008'

In [5]: 'Linux' in uname
Out[5]: True

In [6]: 'Darwin' in uname
Out[6]: False

In [7]: 'Linux' not in uname
Out[7]: False

In [8]: 'Darwin' not in uname
Out[8]: True
```

Если строка string2 содержит строку string1, то выражение string1 in string2 вернет значение True, в противном случае – значение False. Поэтому проверка вхождения строки "Linux" в строку uname в нашем случае дает в результате значение True, а проверка вхождения строки "Darwin" в строку uname дает значение False. Применение оператора not in мы привели «для комплекта».

Иногда бывает достаточно узнать, что некоторая строка является подстрокой другой строки. А иногда требуется узнать, в какой позиции находится искомая подстрока. Выяснить это можно с помощью методов `find()` и `index()`, как показано в примере 3.5.

Пример 3.5. Методы find( ) и index( )

```
In [9]: uname.index('Linux')
Out[9]: 0

In [10]: uname.find('Linux')
Out[10]: 0

In [11]: uname.index('Darwin')
---------------------------------------------------------------------------
<type 'exceptions.ValueError'>            Traceback (most recent call last)

/home/jmjones/code/<ipython console> in <module>()

<type 'exceptions.ValueError'>: substring not found

In [12]: uname.find('Darwin')
Out[12]: -1
```

Если строка `string1` присутствует в строке `string2` (как в данном примере), метод `string2.find(string1)` вернет индекс первого символа `string1` в строке `string2`, в противном случае он вернет `-1`. (Не беспокойтесь, к индексам мы перейдем через мгновение). Точно так же, если строка `string1` присутствует в строке `string2`, метод `string2.index(string1)` вернет индекс первого символа `string1` в строке `string2`, в противном случае он возбудит исключение `ValueError`. В данном примере метод `find()` обнаружил подстроку `"Linux"` в начале строки, поэтому он вернул значение `0`. Однако метод `find()` не смог обнаружить подстроку `"Darwin"` в этой строке, поэтому он вернул значение `-1`. Когда в операционной системе Linux была выполнена попытка отыскать подстроку `"Linux"` с помощью метода `index()`, был получен тот же результат, что и в случае применения метода `find()`. Но при попытке отыскать подстроку `"Darwin"` метод `index()` возбудил исключение `ValueError`, показывая, что не смог отыскать эту подстроку.

Итак, что можно делать с этими числовыми «индексами»? Зачем они нам нужны? Строки интерпретируются как списки символов. «Индекс», который возвращается методами `find()` и `index()`, просто показывает, начиная с какого символа в большей строке было обнаружено совпадение, как показано в примере 3.6.

Пример 3.6. Срез строки

```
In [13]: smp_index = uname.index('SMP')

In [14]: smp_index

Out[14]: 9

In [15]: uname[smp_index:]
```

```
Out[15]: 'SMP Tue Feb 12 02:46:46 UTC 2008'

In [16]: uname[:smp_index]

Out[16]: 'Linux #1 '

In [17]: uname

Out[17]: 'Linux #1 SMP Tue Feb 12 02:46:46 UTC 2008'
```

Мы оказались в состоянии увидеть все символы, начиная с символа, индекс которого был получен в результате поиска подстроки `"SMP"`, и до конца строки, воспользовавшись синтаксической конструкцией извлечения среза `string[index:]`. Мы также смогли увидеть все символы от начала строки `uname` до индекса, который был получен в результате поиска подстроки `"SMP"`, применив синтаксическую конструкцию извлечения среза `string[:index]`. Все различия между этими двумя конструкциями заключаются в местоположении символа двоеточия (`:`) относительно индекса.

Цель примеров на извлечение среза строки и применения операторов `in` и `not in` состоит в том, чтобы показать вам, что строки являются последовательностями и поэтому обладают теми же особенностями, что и другие последовательности, такие как списки. Более полно последовательности обсуждаются в разделе «Sequence Operations» в главе 4 книги «Python in a Nutshell» (издательство O'Reilly) Алекса Мартелли (Alex Martelli) (этот раздел доступен в Интернете на сайте издательства: *http://safari.oreilly.com/0596100469/pythonian-CHP-4-SECT-6*).

Еще два строковых метода, `startswith()` и `endswith()`, как следует из их названий, помогут определить, «начинается» ли или «заканчивается» ли строка определенной подстрокой, как показано в примере 3.7.

Пример 3.7. Методы startswith( ) и endswith( )

```
In [1]: some_string = "Raymond Luxury-Yacht"

In [2]: some_string.startswith("Raymond")
Out[2]: True

In [3]: some_string.startswith("Throatwarbler")
Out[3]: False

In [4]: some_string.endswith("Luxury-Yacht")
Out[4]: True

In [5]: some_string.endswith("Mangrove")
Out[5]: False
```

Как видите, интерпретатор Python возвращает информацию, которая говорит о том, что строка «Raymond Luxury-Yacht» начинается с подстроки «Raymond» и заканчивается подстрокой «Luxury-Yacht». Она не начинается с подстроки «Throatwarbler» и не заканчивается подстрокой «Mangrove». Достаточно просто те же результаты можно получить

с помощью операции извлечения среза, но такой подход к решению выглядит менее наглядно и может показаться несколько утомительным в реализации, как показано в примере 3.8:

Пример 3.8. Имитация методов startswith( ) и endswith( )

```
In [6]: some_string[:len("Raymond")] == "Raymond"
Out[6]: True

In [7]: some_string[:len("Throatwarbler")] == "Throatwarbler"
Out[7]: False

In [8]: some_string[-len("Luxury-Yacht"):] == "Luxury-Yacht"
Out[8]: True

In [9]: some_string[-len("Mangrove"):] == "Mangrove"
Out[9]: False
```

Операция извлечения среза создает и возвращает новый строковый объект, а не изменяет саму строку. Если операции извлечения среза часто используются в сценарии, они могут оказывать существенное влияние на потребление памяти и на производительность. Даже если заметного влияния на производительность не ощущается, тем не менее, лучше воздержаться от использования операций извлечения среза в случаях, когда достаточно применения методов `startswith()` и `endswith()`.

Мы сумели убедиться, что первые символы в строке `some_string`, число которых равно длине строки «Raymond», соответствуют строке «Raymond». Другими словами, мы сумели убедиться, что строка `some_string` начинается с подстроки «Raymond», без использования метода `startswith()`. Точно так же мы смогли убедиться, что строка заканчивается подстрокой «Luxury-Yacht».

Методы `lstrip()`, `rstrip()` и `strip()` без аргументов удаляют ведущие, заключительные, и ведущие и заключительные пробельные символы, соответственно. В качестве таких пробельных символов можно назвать символы табуляции, пробелы, символы возврата каретки и новой строки. Метод `lstrip()` без аргументов удаляет любые пробельные символы, которые находятся в начале строки, и возвращает новую строку. Метод `rstrip()` без аргументов удаляет любые пробельные символы, которые находятся в конце строки, и возвращает новую строку. Метод `strip()` без аргументов удаляет любые пробельные символы, которые находятся в начале и в конце строки, и возвращает новую строку, как показано в примере 3.9.

Все три метода из семейства `strip()` не изменяют саму строку, а создают и возвращают новый строковый объект. Возможно, вы никогда не будете испытывать проблем с таким поведением методов, но вы должны знать о нем.

Пример 3.9. Методы lstrip( ), rstrip( ) и strip( )

```
In [1]: spacious_string = "\n\t Some Non-Spacious Text\n \t\r"

In [2]: spacious_string
Out[2]: '\n\t Some Non-Spacious Text\n \t\r'

In [3]: print spacious_string

         Some Non-Spacious Text

In [4]: spacious_string.lstrip()
Out[4]: 'Some Non-Spacious Text\n \t\r'

In [5]: print spacious_string.lstrip()
Some Non-Spacious Text


In [6]: spacious_string.rstrip()
Out[6]: '\n\t Some Non-Spacious Text'

In [7]: print spacious_string.rstrip()

         Some Non-Spacious Text

In [8]: spacious_string.strip()
Out[8]: 'Some Non-Spacious Text'

In [9]: print spacious_string.strip()
Some Non-Spacious Text
```

Все три метода, `lstrip()`, `rstrip()` и `strip()`, могут принимать единственный необязательный аргумент: строку символов, которые следует удалить из соответствующего места строки. Это означает, что методы семейства `strip()` не просто удаляют пробельные символы – они могут удалять любые символы, какие вы укажете:

```
In [1]: xml_tag = "<some_tag>"

In [2]: xml_tag.lstrip("<")

Out[2]: 'some_tag>'

In [3]: xml_tag.lstrip(">")

Out[3]: '<some_tag>'

In [4]: xml_tag.rstrip(">")

Out[4]: '<some_tag'

In [5]: xml_tag.rstrip("<")

Out[5]: '<some_tag>'
```

Здесь мы удалили из тега XML левую и правую угловые скобки, по одной за раз. А как быть, если нам потребуется удалить обе скобки одновременно? Сделать это можно следующим способом:

```
In [6]: xml_tag.strip("<").strip(">")

Out[6]: 'some_tag'
```

Методы семейства `strip()` возвращают строку, поэтому мы можем вызывать другие строковые операции прямо вслед за вызовом метода `strip()`. В этом примере мы объединили вызовы методов `strip()` в цепочку. Первый вызов метода `strip()` удаляет начальный символ (левую угловую скобку) и возвращает строку, а второй метод `strip()` удаляет завершающий символ (правую угловую скобку) и возвращает строку `"some_tag"`. Однако существует более простой способ:

```
In [7]: xml_tag.strip("<>")

Out[7]: 'some_tag'
```

Возможно, вы подумали, что методы семейства `strip()` удаляют точное вхождение указанной подстроки, но в действительности удаляются любые последовательные вхождения указанных символов с соответствующей стороны строки. В этом последнем примере методу `strip()` было предписано удалить `"<>"`. Это не означает точное соответствие подстроке `"<>"` и не означает, что должны быть удалены вхождения этих двух символов, следующих друг за другом именно в таком порядке, – это означает, что должны быть удалены символы `"<"` или `">"`, находящиеся в начале или в конце строки.

Ниже приводится, возможно, более понятный пример:

```
In [8]: gt_lt_str = "<><><>gt lt str<><><>"

In [9]: gt_lt_str.strip("<>")

Out[9]: 'gt lt str'

In [10]: gt_lt_str.strip("><")

Out[10]: 'gt lt str'
```

Здесь мы удалили все вхождения символов `"<"` или `">"` с обоих концов строки. Таким способом мы можем ликвидировать простые символы и пробелы.

Следует заметить, что такой прием может работать несколько не так, как вы ожидаете, например:

```
In [11]: foo_str = "<fooooooo>blah<foo>"

In [12]: foo_str.strip("<foo>")

Out[12]: 'blah'
```

У вас могло бы сложиться мнение, что метод `strip()` в этом примере удалит символы справа, но не слева. Однако он обнаружит и удалит любые последовательные вхождения символов `"<"`, `"f"`, `"o"` и `">"`. Это не ошибка, мы не пропустили второй символ `"o"`. Вот еще один, заключительный пример использования метода `strip()`, который прояснит это утверждение:

```
In [13]: foo_str.strip(">><of")

Out[13]: 'blah'
```

Здесь удаляются символы ">", "<", "f", "o", хотя они следуют не в этом порядке.

Методы upper() и lower() удобно использовать, когда необходимо выполнить сравнение двух строк без учета регистра символов. Метод upper() возвращает строку со всеми символами из оригинальной строки в верхнем регистре. Метод lower() возвращает строку со всеми символами из оригинальной строки в нижнем регистре, как показано в примере 3.10.

Пример 3.10. Методы upper( ) и lower( )

```
In [1]: mixed_case_string = "VOrpal BUnny"

In [2]: mixed_case_string == "vorpal bunny"
Out[2]: False

In [3]: mixed_case_string.lower() == "vorpal bunny"
Out[3]: True

In [4]: mixed_case_string == "VORPAL BUNNY"
Out[4]: False

In [5]: mixed_case_string.upper() == "VORPAL BUNNY"
Out[5]: True

In [6]: mixed_case_string.upper()
Out[6]: 'VORPAL BUNNY'

In [7]: mixed_case_string.lower()
Out[7]: 'vorpal bunny'
```

Если вам необходимо извлечь часть строки, ограниченной какими-либо символами-разделителями, метод split() предоставит вам эту возможность, как показано в примере 3.11.

Пример 3.11. Метод split( )

```
In [1]: comma_delim_string = "pos1,pos2,pos3"

In [2]: pipe_delim_string = "pipepos1|pipepos2|pipepos3"

In [3]: comma_delim_string.split(',')
Out[3]: ['pos1', 'pos2', 'pos3']

In [4]: pipe_delim_string.split('|')
Out[4]: ['pipepos1', 'pipepos2', 'pipepos3']
```

Методу split() передается строка-разделитель, по которому необходимо разбить строку на подстроки. Часто это единственный символ, такой как запятая или вертикальная черта, но это может быть строка, содержащая более одного символа. В данном примере мы разбили

строку comma_delim_string по запятым, а строку pipe_delim_string — по символу вертикальной черты (|), передавая символ запятой или вертикальной черты методу split(). Возвращаемым значением метода является список строк, каждая из которых представляет собой группу последовательных символов, находящихся между разделителями. Когда в качестве разделителя необходимо использовать не единственный символ, а некоторую строку, метод split() справится и с этим. К моменту написания этих строк в языке Python отсутствовал символьный тип, поэтому хотя в примерах метод split() получал единственный символ, он рассматривался методом как строка. Поэтому, когда методу split() передается несколько символов, он обработает и их, как показано в примере 3.12.

Пример 3.12. Строка-разделитель в методе split( )

```
In [1]: multi_delim_string = "pos1XXXpos2XXXpos3"

In [2]: multi_delim_string.split("XXX")
Out[2]: ['pos1', 'pos2', 'pos3']

In [3]: multi_delim_string.split("XX")
Out[3]: ['pos1', 'Xpos2', 'Xpos3']

In [4]: multi_delim_string.split("X")
Out[4]: ['pos1', '', '', 'pos2', '', '', 'pos3']
```

Обратите внимание, что сначала мы использовали в качестве разделителя строку "XXX" для разделения строки multi_delim_string. Как и ожидалось, в результате был получен список ['pos1', 'pos2', 'pos3']. Затем, мы использовали в качестве разделителя строку "XX" и метод split() вернул ['pos1', 'Xpos2', 'Xpos3']. Здесь метод split() выбрал все символы, находящиеся между соседними разделителями "XX". Подстрока "pos1" начинается с начала строки и простирается до первого разделителя "XX"; подстрока "Xpos2" располагается сразу за первым вхождением разделителя "XX" и простирается до второго его вхождения; и подстрока "Xpos3" располагается сразу за вторым вхождением "XX" и простирается до конца строки. Последний вызов метода split() получает в качестве разделителя единственный символ "X". Обратите внимание, что позициям между соседними символами "X" соответствуют пустые строки ("") в результирующем списке. Это означает, что между соседними символами "X" ничего нет.

Но как быть, если необходимо разбить строку только по первым n вхождениям указанного разделителя? Для этого методу split() следует передать второй аргумент, с именем max_split. Когда во втором аргументе max_split методу split() передается целочисленное значение, он выполнит только указанное число разбиений исходной строки:

```
In [1]: two_field_string = "8675309,This is a freeform, plain text, string"

In [2]: two_field_string.split(',', 1)
Out[2]: ['8675309', 'This is a freeform, plain text, string']
```

Здесь мы разбиваем строку по запятым и предписываем методу `split()` выполнить только одно разбиение. Несмотря на то, что в строке присутствует несколько запятых, строка была разбита только один раз.

Если необходимо разбить строку по пробелам, например, чтобы извлечь из текста отдельные слова, эту задачу легко можно решить вызовом метода `split()` без аргументов:

```
In [1]: prosaic_string = "Insert your clever little piece of text here."

In [2]: prosaic_string.split()
Out[2]: ['Insert', 'your', 'clever', 'little', 'piece', 'of', 'text', 'here.']
```

Когда метод `split()` не получает никаких аргументов, по умолчанию он выполняет разбиение строки по пробельным символам.

Чаще всего вы будете получать именно те результаты, которые ожидали получить. Однако в случае многострочного текста результат может получиться неожиданным для вас. Часто при работе с многострочным текстом бывает необходимо выполнять его обработку по одной строке за раз. Но вы можете с удивлением обнаружить, что программа разбивает такой текст на отдельные слова:

```
In [1]: multiline_string = """This
   ...: is
   ...: a multiline
   ...: piece of
   ...: text"""

In [2]: multiline_string.split()
Out[2]: ['This', 'is', 'a', 'multiline', 'piece', 'of', 'text']
```

Для таких случаев лучше подходит метод `splitlines()`:

```
In [3]: lines = multiline_string.splitlines()

In [4]: lines

Out[4]: ['This', 'is', 'a multiline', 'piece of', 'text']
```

Метод `splitlines()` возвращает список всех строк из многострочного текста и сохраняет группы «слов». После этого можно выполнить итерации по отдельным строкам текста и извлечь отдельные слова:

```
In [5]: for line in lines:
   ...:     print "START LINE::"
   ...:     print line.split()
   ...:     print "::END LINE"
   ...:
START LINE::
['This']
::END LINE
START LINE::
['is']
::END LINE
```

```
START LINE::
['a', 'multiline']
::END LINE
START LINE::
['piece', 'of']
::END LINE
START LINE::
['text']
::END LINE
```

Иногда бывает необходимо не анализировать строку или извлекать из нее информацию, а объединить в строку уже имеющиеся данные. В этом случае вам на помощь придет метод `join()`:

```
In [1]: some_list = ['one', 'two', 'three', 'four']

In [2]: ','.join(some_list)
Out[2]: 'one,two,three,four'

In [3]: ', '.join(some_list)
Out[3]: 'one, two, three, four'

In [4]: '\t'.join(some_list)
Out[4]: 'one\ttwo\tthree\tfour'

In [5]: ''.join(some_list)
Out[5]: 'onetwothreefour'
```

Учитывая, что исходные данные хранятся в виде списка, мы можем объединить строки `'one'`, `'two'`, `'three'` и `'four'` несколькими способами. Мы объединяем элементы списка `some_list` с помощью запятой, запятой и пробела, символа табуляции и пустой строки. Метод `join()` – это строковый метод, поэтому вызов его в качестве метода литерала, такого как `','`, является корректным. Метод `join()` принимает в качестве аргумента последовательность строк и объединяет их в одну строку так, чтобы элементы последовательности располагались в исходном порядке и отделялись строкой, для которой вызывается метод `join()`.

Мы должны предупредить вас об особенностях поведения метода `join()` и об аргументе, который он ожидает получить. Обратите внимание: метод `join()` ожидает получить последовательность строк. А что произойдет, если ему передать последовательность целых чисел? Взгляните!

```
In [1]: some_list = range(10)

In [2]: some_list
Out[2]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

In [3]: ",".join(some_list)
---------------------------------------------------------------------------
exceptions.TypeError                      Traceback (most recent call last)

/Users/jmjones/<ipython console>
```

```
TypeError: sequence item 0: expected string, int found
(TypeError: элемент 0 последовательности: ожидается строка, обнаружено целое)
```

Диагностическая информация, приложенная к исключению, возбужденному методом `join()`, достаточно ясно объясняет происшедшее, но так как это довольно распространенная ошибка, в этом стоит разобраться. Вы легко сможете избежать этой ловушки с помощью простого генератора списков. Ниже мы прибегли к помощи генератора списков, чтобы преобразовать все элементы списка `some_list`, которые содержат целые числа, в строки:

```
In [4]: ",".join([str(i) for i in some_list])
Out[4]: '0,1,2,3,4,5,6,7,8,9'
```

Или можно использовать выражение-генератор:

```
In [5]: ",".join(str(i) for i in some_list)

Out[5]: '0,1,2,3,4,5,6,7,8,9'
```

За дополнительной информацией об использовании генераторов списков обращайтесь к разделу «Control Flow Statements» в главе 4 книги «Python in a Nutshell» (этот раздел доступен в Интернете, на сайте издательства: *http://safari.oreilly.com/0596100469/pythonian-CHP-4-SECT-10*).

Последний метод, используемый для создания и изменения текстовых строк, – это метод `replace()`. Метод `replace()` принимает два аргумента: строку, которую требуется заменить, и строку замены, соответственно. Ниже приводится простой пример использования метода `replace()`:

```
In [1]: replacable_string = "trancendental hibernational nation"

In [2]: replacable_string.replace("nation", "natty")
Out[2]: 'trancendental hibernattyal natty'
```

Обратите внимание, что метод `replace()` никак не проверяет, является замещаемая строка частью слова или отдельным словом. Поэтому метод `replace()` может использоваться только в случаях, когда просто требуется заменить определенную последовательность символов другой определенной последовательностью символов.

Иногда требуется более тонкое управление операцией замены, когда вариант простой замены одной последовательности символов на другую не подходит. В таких случаях обычно бывает необходимо иметь возможность определить шаблон последовательности символов, которую требуется найти и заменить. Применение шаблонов также может помочь с поиском требуемого текста для последующего извлечения из него данных. В тех случаях, когда предпочтительнее использовать шаблоны, вам помогут регулярные выражения. Регулярные выражения мы рассмотрим далее.

Так же, как операция извлечения среза и метод `strip()`, метод `replace()` не изменяет существующую строку, а создает новую.

Строки Юникода

До сих пор во всех примерах работы со строками, которые мы видели, использовались исключительно строковые объекты встроенного типа `str`, но в языке Python существует еще один строковый тип, с которым вам предстоит познакомиться: строки Юникода. Любые символы, которые выводятся на экран дисплея, внутри компьютера представлены числами. До появления кодировки Юникод существовало множество разнообразных наборов отображения числовых кодов в символы в зависимости от языка и платформы. Юникод – это стандарт, обеспечивающий единое отображение числовых кодов в символы, независимое от языка, платформы или даже программы, выполняющей обработку текста. В этом разделе мы рассмотрим понятие Юникода и способы работы с этой кодировкой, имеющиеся в языке Python. Подробное описание Юникода вы найдете в превосходном учебнике Э. М. Качлинга (A. M. Kuchling) по адресу: *http://www.amk.ca/python/howto/unicode*.

Создание строк Юникода выглядит ничуть не сложнее, чем создание обычных строк:

```
In [1]: unicode_string = u'this is a unicode string'

In [2]: unicode_string
Out[2]: u'this is a unicode string'

In [3]: print unicode_string
this is a unicode string
```

Или можно воспользоваться встроенной функцией `unicode()`:

```
In [4]: unicode('this is a unicode string')
Out[4]: u'this is a unicode string'
```

На первый взгляд, в этом нет ничего примечательного, особенно если учесть, что здесь мы имеем дело с символами одного языка. Но как быть, когда приходится работать с символами из нескольких языков? Здесь вам на помощь придет Юникод. Чтобы внутри строки Юникода создать символ с определенным числовым кодом, можно воспользоваться нотацией `\uXXXX` или `\uXXXXXXXX`. Например, ниже приводится строка Юникода, содержащая символы латиницы, греческого алфавита и кириллицы:

```
In [1]: unicode_string = u'abc_\u03a0\u03a3\u03a9_\u0414\u0424\u042F'

In [2]: unicode_string
Out[2]: u'abc_\u03a0\u03a3\u03a9_\u0414\u0424\u042f'
```

Интерпретатор генерирует строку (`str`) в зависимости от используемой кодировки. В версии Python, которая поставляется вместе с компьютерами Mac, если попытаться вывести строку из предыдущего примера с помощью инструкции `print`, будет получено сообщение об ошибке:

```
In [3]: print unicode_string
---------------------------------------------------------------------------
UnicodeEncodeError                        Traceback (most recent call last)

/Users/jmjones/<ipython console> in <module>()

UnicodeEncodeError: 'ascii' codec can't encode characters in position 4-6:
  ordinal not in range(128)
(UnicodeEncodeError: кодек 'ascii' не в состоянии кодировать символы
в позиции 4-6: числовые значения находятся вне диапазона range(128))
```

Мы должны определить другой кодек, который знает, как обрабатывать все символы в строке:

```
In [4]: print unicode_string.encode('utf-8')
abc_ΠΣΩ_ДФЯ
```

Здесь мы выполнили кодирование строки, содержащей символы латиницы, греческого алфавита и кириллицы, в кодировку UTF-8, которая наиболее часто используется для кодирования данных Юникода.

Строки Юникода обладают теми же возможностями, такими как возможность выполнения проверки с помощью оператора `in`, и методами, что и обычные строки, о которых мы уже говорили:

```
In [5]: u'abc' in unicode_string
Out[5]: True

In [6]: u'foo' in unicode_string
Out[6]: False

In [7]: unicode_string.split()
Out[7]: [u'abc_\u03a0\u03a3\u03a9_\u0414\u0424\u042f']

In [8]: unicode_string.
unicode_string.__add__           unicode_string.expandtabs
unicode_string.__class__         unicode_string.find
unicode_string.__contains__      unicode_string.index
unicode_string.__delattr__       unicode_string.isalnum
unicode_string.__doc__           unicode_string.isalpha
unicode_string.__eq__            unicode_string.isdecimal
unicode_string.__ge__            unicode_string.isdigit
unicode_string.__getattribute__  unicode_string.islower
unicode_string.__getitem__       unicode_string.isnumeric
unicode_string.__getnewargs__    unicode_string.isspace
unicode_string.__getslice__      unicode_string.istitle
unicode_string.__gt__            unicode_string.isupper
unicode_string.__hash__          unicode_string.join
unicode_string.__init__          unicode_string.ljust
unicode_string.__le__            unicode_string.lower
```

```
unicode_string.__len__          unicode_string.lstrip
unicode_string.__lt__           unicode_string.partition
unicode_string.__mod__          unicode_string.replace
unicode_string.__mul__          unicode_string.rfind
unicode_string.__ne__           unicode_string.rindex
unicode_string.__new__          unicode_string.rjust
unicode_string.__reduce__       unicode_string.rpartition
unicode_string.__reduce_ex__    unicode_string.rsplit
unicode_string.__repr__         unicode_string.rstrip
unicode_string.__rmod__         unicode_string.split
unicode_string.__rmul__         unicode_string.splitlines
unicode_string.__setattr__      unicode_string.startswith
unicode_string.__str__          unicode_string.strip
unicode_string.capitalize       unicode_string.swapcase
unicode_string.center           unicode_string.title
unicode_string.count            unicode_string.translate
unicode_string.decode           unicode_string.upper
unicode_string.encode           unicode_string.zfill
unicode_string.endswith
```

Возможно, строки Юникода не потребуются вам немедленно. Но важно знать об их существовании, если вы собираетесь продолжать программировать на языке Python.

re

Раз поставка языка Python комплектуется в соответствии с принципом «батарейки включены», можно было бы ожидать, что в состав стандартной библиотеки будут включены модули для работы с регулярными выражениями. Так оно и есть. Акцент в этом разделе сделан на использовании в языке Python регулярных выражений, а не на подробностях их синтаксиса. Поэтому, если вы не знакомы с регулярными выражениями, рекомендуем вам приобрести книгу «Mastering Regular Expressions» (O'Reilly) Джеффри Е. Ф. Фридла (Jeffrey E. F. Friedl) (доступна также в Интернете на сайте издательства по адресу: *http://safari.oreilly.com/0596528124*).[1] Далее мы предполагаем, что вы достаточно уверенно оперируете регулярными выражениями, в противном случае рекомендуем держать книгу Фридла под рукой.

Если вы знакомы с языком Perl, то, возможно, вы уже использовали регулярные выражения с оператором `=~`. В языке Python поддержка регулярных выражений реализована на уровне библиотеки, а не на уровне синтаксических особенностей языка. Поэтому для работы с регулярными выражениями необходимо импортировать модуль `re`. Ни-

[1] Джеффри Фридл «Регулярные выражения», 3-е издание. – Пер. с англ. – СПб.: Символ-плюс, 2008. В Интернете можно ознакомиться с 5-й главой этой книги по адресу *http://www.books.ru/chapter?id= 592346&num=1*. – *Прим. перев.*

же приводится простой пример создания и использования регулярного выражения, как показано в примере 3.13.

Пример 3.13. Простой пример использования регулярного выражения

```
In [1]: import re

In [2]: re_string = "{{(.*?)}}"

In [3]: some_string = "this is a string with {{words}} embedded in\
   ...: {{curly brackets}} to show an {{example}} of {{regular expressions}}"

In [4]: for match in re.findall(re_string, some_string):
   ...:     print "MATCH->", match
   ...:
MATCH-> words
MATCH-> curly brackets
MATCH-> example
MATCH-> regular expressions
```

Первое, что мы сделали в этом примере, – импортировали модуль `re`. Как вы уже наверняка поняли, имя `re` происходит от «regular expressions» (регулярные выражения). Затем мы создали строку `re_string`, которая будет играть роль шаблона для поиска. Этому шаблону будут соответствовать две открывающие фигурные скобки (`{{`), вслед за которыми может следовать текст, завершающийся двумя закрывающими фигурными скобками (`}}`). Затем мы создали строку `some_string`, которая содержит группы слов, окруженные фигурными скобками. И в конце мы выполнили обход результатов поиска в строке `some_string` по шаблону `re_string`, полученных от функции `findall()` из модуля `re`. Как видите, пример вывел строки `words`, `curly brackets`, `example` и `regular expressions`, которые представляют все группы слов, заключенные в двойные фигурные скобки.

В языке Python существует два способа работы с регулярными выражениями. Первый заключается в непосредственном использовании функций из модуля `re`, как в предыдущем примере. Второй способ состоит в том, чтобы создать объект скомпилированного регулярного выражения и затем использовать методы этого объекта.

Итак, что же такое скомпилированное регулярное выражение? Это просто объект, созданный вызовом функции `re.compile()`, которой передается шаблон. Этот объект, созданный за счет передачи шаблона функции `re.compile()`, содержит множество методов для работы с регулярным выражением. Между скомпилированным и нескомпилированным регулярными выражениями имеются два основных отличия. Во-первых, вместо ссылки на шаблон регулярного выражения `"{{.*?}}"` создается объект скомпилированного выражения на основе шаблона. Во-вторых, вместо функции `findall()` из модуля `re` следует вызывать метод `findall()` объекта скомпилированного выражения.

За дополнительной информацией о всех функциях, имеющихся в модуле re, обращайтесь к разделу «Module Contents» в справочнике «Python Library Reference», *http://docs.python.org/lib/node46.html*. За дополнительной информацией об объектах скомпилированных регулярных выражений обращайтесь к разделу «Regular Expression Objects» в справочнике «Python Library Reference», *http://docs.python.org/lib/re-objects.html*.

В примере 3.14 представлена реализация предыдущего примера с двойными фигурными скобками, выполненная на основе использования объекта скомпилированного регулярного выражения.

Пример 3.14. Простое регулярное выражение, скомпилированный шаблон

```
In [1]: import re

In [2]: re_obj = re.compile("{{(.*?)}}")

In [3]: some_string = "this is a string with {{words}} embedded in\
   ...: {{curly brackets}} to show an {{example}} of {{regular expressions}}"

In [4]: for match in re_obj.findall(some_string):
   ...:     print "MATCH->", match
   ...:
MATCH-> words
MATCH-> curly brackets
MATCH-> example
MATCH-> regular expressions
```

Выбор метода работы с регулярными выражениями в языке Python зависит отчасти от личных предпочтений и от самого регулярного выражения. Следует заметить, что метод, основанный на использовании функций модуля re, уступает в производительности методу, основанному на использовании объектов скомпилированных регулярных выражений. Проблема производительности особенно остро может вставать, например, когда регулярное выражение применяется в цикле к каждой строке текстового файла, содержащего сотни и тысячи строк. В примерах ниже представлены реализации простых сценариев, использующих скомпилированые и нескомпилированные регулярные выражения, которые применяются к файлу, содержащему 500 000 строк текста. Если воспользоваться специальной функцией timeit, можно увидеть разницу в производительности между этими двумя сценариями. Смотрите пример 3.15.

Пример 3.15. Тест производительности нескомпилированного регулярного выражения

```
#!/usr/bin/env python

import re

def run_re():
    pattern = 'pDq'
```

```
    infile = open('large_re_file.txt', 'r')
    match_count = 0
    lines = 0
    for line in infile:
        match = re.search(pattern, line)
        if match:
            match_count += 1
        lines += 1
    return (lines, match_count)

if __name__ == "__main__":
    lines, match_count = run_re()
    print 'LINES::', lines
    print 'MATCHES::', match_count
```

Функция `timeit` выполняет программный код несколько раз и возвращает время самого лучшего варианта. Ниже показаны результаты запуска утилиты `timeit` для этого сценария в оболочке IPython:

```
In [1]: import re_loop_nocompile

In [2]: timeit -n 5 re_loop_nocompile.run_re()
5 loops, best of 3: 1.93 s per loop
```

В этом примере функция `run_re()` была вызвана 5 раз, и было вычислено среднее из 3 самых лучших показателей, которое составило 1,93 секунды. Специальная функция `timeit` выполняется с исследуемым программным кодом несколько раз, чтобы уменьшить погрешность, вызванную влиянием других процессов, исполняющихся в системе.

Ниже приводятся результаты измерения времени выполнения того же самого программного кода с помощью утилиты `time` операционной системы UNIX:

```
jmjones@dink:~/code$ time python re_loop_nocompile.py
LINES:: 500000 MATCHES:: 242

real    0m2.113s
user    0m1.888s
sys     0m0.163s
```

Пример 3.16 – это тот же пример с регулярным выражением за исключением того, что мы используем `re.compile()` для создания объекта скомпилированного шаблона.

Пример 3.16. Тест производительности скомпилированного регулярного выражения

```
#!/usr/bin/env python

import re

def run_re():
    pattern = 'pDq'
    re_obj = re.compile(pattern)
```

```
    infile = open('large_re_file.txt', 'r')
    match_count = 0
    lines = 0
    for line in infile:
        match = re_obj.search(line)
        if match:
            match_count += 1
        lines += 1
    return (lines, match_count)

if __name__ == "__main__":
    lines, match_count = run_re()
    print 'LINES::', lines
    print 'MATCHES::', match_count
```

Испытания с помощью специальной функции `timeit` в оболочке IPython дали следующие результаты:

```
In [3]: import re_loop_compile

In [4]: timeit -n 5 re_loop_compile.run_re()
5 loops, best of 3: 860 ms per loop
```

А испытания того же самого сценария с помощью утилиты `time` операционной системы UNIX дали следующие результаты:

```
jmjones@dink:~/code$ time python
re_loop_compile.py LINES:: 500000 MATCHES:: 242

real    0m0.996s
user    0m0.836s
sys     0m0.154s
```

Версия со скомпилированным регулярным выражением одержала чистую победу. Время работы этой версии оказалось в два раза меньше как по данным утилиты UNIX `time`, так и по данным функции `timeit` оболочки IPython. Поэтому мы настоятельно рекомендуем взять в привычку использовать объекты скомпилированных регулярных выражений.

Как уже говорилось ранее в этой главе, для определения строк, в которых не интерпретируются экранированные последовательности, можно использовать сырые (неформатированные) строки. В примере 3.17 показано применение неформатированных строк для использования в регулярных выражениях.

Пример 3.17. Неформатированные строки и регулярные выражения

```
In [1]: import re

In [2]: raw_pattern = r'\b[a-z]+\b'

In [3]: non_raw_pattern = '\b[a-z]+\b'

In [4]: some_string = 'a few little words'
```

```
In [5]: re.findall(raw_pattern, some_string)

Out[5]: ['a', 'few', 'little', 'words']

In [6]: re.findall(non_raw_pattern, some_string)

Out[6]: []
```

Шаблонный символ `\b` в регулярных выражениях соответствует границе слова. То есть, как в случае применения сырой строки, так и в случае применения обычной строки, мы предполагаем отыскать отдельные слова, состоящие из символов нижнего регистра. Обратите внимание, что при использовании `raw_pattern` были обнаружены отдельные слова в `some_string`, а при использовании `non_raw_pattern` вообще ничего не было найдено. В строке `raw_pattern` комбинация `\b` интерпретируется как два отдельных символа, в то время как в строке `non_raw_pattern` она интерпретируется как символ забоя (backspace). В результате функция `findall()` сумела отыскать отдельные слова с помощью неформатированной строки шаблона. Однако при использовании шаблона в виде обычной строки функция `findall()` не отыскала ни одного символа забоя (backspace).

Чтобы с помощью шаблона `non_raw_pattern` можно было отыскать соответствие в строке, необходимо окружить требуемое слово символами `\b`, как показано ниже:

```
In [7]: some_other_string = 'a few \blittle\b words'

In [8]: re.findall(non_raw_pattern, some_other_string)
Out[8]: ['\x08little\x08']
```

Обратите внимание на шестнадцатеричную форму записи символа `"\x08"` в соответствии, найденном функцией `findall()`. Эта шестнадцатеричная форма записи соответствует символам забоя (backspace), которые были добавлены с помощью экранированной последовательности `\b`.

Как видите, неформатированные строки могут пригодиться, когда предполагается использовать специальные последовательности, такие как `"\b"`, обозначающую границу слова, `"\d"`, обозначающую цифру, или `"\w"`, обозначающую алфавитно-цифровой символ. Полный перечень специальных последовательностей, начинающихся с символа обратного слеша, вы найдете в разделе «Regular Expression Syntax» в справочнике «Python Library Reference», *http://docs.python.org/lib/re-syntax.html*.

Примеры с 3.14 по 3.17 были очень простыми. В них во всех использовались регулярные выражения и различные методы, применяемые к ним. Иногда такого ограниченного использования регулярных выражений вполне достаточно. Иногда бывает необходимо нечто более мощное, чем имеется в библиотеке регулярных выражений.

К основным методам (или функциям) регулярных выражений, которые используются наиболее часто, относятся `findall()`, `finditer()`, `match()` и `search()`. Вам также могут потребоваться методы `split()` и `sub()`, но, вероятно, не так часто, как другие методы.

Метод `findall()` отыскивает все вхождения указанного шаблона в строке. Если метод `findall()` найдет соответствия шаблону, тип возвращаемой структуры данных будет зависеть от наличия групп в шаблоне.

Краткое напоминание: группировка в регулярных выражениях позволяет указывать текст внутри регулярного выражения, который следует извлечь из результата. За дополнительной информацией обращайтесь к разделу «Common Metacharacters and Fields» в книге Фридла (Friedl) «Mastering Regular Expressions»[1] или в Интернете по адресу: *http://safari.oreilly.com/0596528124/regex3-CHP-3-SECT-5?imagepage=137*.

Если в регулярном выражении отсутствуют группы, а совпадение найдено, тогда `findall()` вернет список строк. Например:

```
In [1]: import re

In [2]: re_obj = re.compile(r'\bt.*?e\b')

In [3]: re_obj.findall("time tame tune tint tire")
Out[3]: ['time', 'tame', 'tune', 'tint tire']
```

В этом шаблоне отсутствуют группы, поэтому `findall()` возвращает список строк. Здесь можно наблюдать интересный побочный эффект – последний элемент списка содержит два слова, tint и tire. Используемое здесь регулярное выражение соответствует словам, начинающимся с символа «t» и заканчивающимся символом «e». Но часть выражения `.*?` соответствует любым символам, включая пробелы. Метод `findall()` отыскал все, что предполагалось. Он отыскал слово, начинающееся с символа «t» (tint), и продолжил просмотр строки, пока не обнаружил слово, завершающееся символом «e» (tire). Поэтому соответствие «tint tire» вполне согласуется с шаблоном. Чтобы исключить пробел, можно было бы использовать регулярное выражение `r'\bt\w*e\b'`:

```
In [4]: re_obj = re.compile(r'\bt\w*e\b')

In [5]: re_obj.findall("time tame tune tint tire")
Out[5]: ['time', 'tame', 'tune', 'tire']
```

Второй тип структуры данных, который может быть получен, – это список кортежей. Если группы присутствуют в выражении и было найдено совпадение, то `findall()` вернет список кортежей. Подобный шаблон и строка показаны в примере 3.18.

---

[1] Джеффри Фридл «Регулярные выражения», 3-е издание. – Пер. с англ. – СПб.: Символ-плюс, 2008. Глава 3. Раздел «Стандартные метасимволы и возможности». – *Прим. перев.*

Пример 3.18. Простая группировка и метод findall( )

```
In [1]: import re

In [2]: re_obj = re.compile(r"""(A\W+\b(big|small)\b\W+\b
   ...: (brown|purple)\b\W+\b(cow|dog)\b\W+\b(ran|jumped)\b\W+\b
   ...: (to|down)\b\W+\b(the)\b\W+\b(street|moon).*?\.)""",
   ...: re.VERBOSE)

In [3]: re_obj.findall('A big brown dog ran down the street. \
   ...: A small purple cow jumped to the moon.')

Out[3]:
[('A big brown dog ran down the street.',
  'big',
  'brown',
  'dog',
  'ran',
  'down',
  'the',
  'street'),
 ('A small purple cow jumped to the moon.',
  'small',
  'purple',
  'cow',
  'jumped',
  'to',
  'the',
  'moon')]
```

Несмотря на свою простоту, этот пример демонстрирует ряд важных моментов. Во-первых, обратите внимание, что этот простой шаблон нелепо длинен и содержит массу неалфавитно-цифровых символов, от которых начинает рябить в глазах, если смотреть слишком долго. Это обычная вещь для многих регулярных выражений. Затем, обратите внимание, что шаблон содержит явные вложенные группы. Объемлющая группа будет соответствовать любому тексту, начинающемуся с символа «A» и заканчивающемуся точкой. Символы между начальным символом «A» и завершающей точкой образуют вложенные группы, которые должны соответствовать словам «big» или «small», «brown» или «purple» и так далее. Далее, возвращаемое значение метода `findall()` представляет собой список кортежей. Элементами этих кортежей являются группы, которые были определены в регулярном выражении. Первый элемент кортежа – все предложение, потому что оно соответствует наибольшей, объемлющей группе. Последующие элементы кортежа соответствуют каждой из подгрупп. Наконец, обратите внимание на последний аргумент в вызове метода `re.compile()` – `re.VERBOSE`. Это позволило нам записать регулярное выражение в многострочном режиме, то есть мы смогли расположить регулярное выражение в нескольких строках, не оказывая влияния на поиск соответствий. Пробел, оказавшийся за пределами группировки, был проигнори-

рован. Хотя мы и не продемонстрировали здесь такую возможность, тем не менее, многострочный режим позволяет вставлять комментарии в конец каждой строки регулярного выражения, чтобы описать, что делает та или иная его часть. Одна из основных сложностей, связанных с регулярными выражениями, состоит в том, что описание шаблона часто бывает очень длинным и трудным для чтения. Цель `re.VERBOSE` состоит в том, чтобы упростить написание регулярных выражений, следовательно, это ценный инструмент, облегчающий сопровождение программного кода, содержащего регулярные выражения.

Метод `finditer()` является разновидностью метода `findall()`. Вместо того чтобы возвращать список кортежей, как это делает метод `findall()`, `finditer()` возвращает итератор, как это следует из имени метода. Каждый элемент итератора – это объект найденного совпадения, который мы обсудим далее в этой главе. Пример 3.19 реализует тот же простой пример, только в нем вместо метода `findall()` используется метод `finditer()`.

Пример 3.19. Пример использования метода finditer( )

```
In [4]: re_iter = re_obj.finditer('A big brown dog ran down the street. \
   ...: A small purple cow jumped to the moon.')

In [5]: re_iter

Out[5]: <callable-iterator object at 0xa17ad0>

In [6]: for item in re_iter:
   ...:     print item
   ...:     print item.groups()
   ...:
<_sre.SRE_Match object at 0x9ff858>
('A big brown dog ran down the street.', 'big', 'brown', 'dog', 'ran',
    'down', 'the', 'street')
<_sre.SRE_Match object at 0x9ff940>
('A small purple cow jumped to the moon.', 'small', 'purple', 'cow',
    'jumped', 'to', 'the', 'moon')
```

Если прежде вы никогда не сталкивались с итераторами, вы можете представлять их себе как списки, которые создаются в тот момент, когда они необходимы. Один из недостатков такого определения состоит в том, что вы не можете обратиться к определенному элементу итератора по его индексу, как, например, к элементу списка `some_list[3]`. Вследствие этого ограничения вы не можете получить срез итератора, как, например, в случае списка `some_list[2:6]`. Тем не менее, независимо от этих ограничений итераторы представляют собой легкое и мощное средство, особенно когда необходимо выполнить итерации через некоторую последовательность, потому что при этом последовательность не загружается целиком в память, а элементы ее возвращаются по требованию. Это позволяет итераторам занимать меньший объем

памяти, чем соответствующие им списки. Кроме того, доступ к элементам последовательности производится быстрее.

Еще одно преимущество метода `finditer()` перед `findall()` состоит в том, что каждый элемент, возвращаемый методом `finditer()`, – это объект `match`, а не простой список строк или список кортежей, соответствующих найденному тексту.

Методы `match()` и `search()` обеспечивают похожие функциональные возможности. Оба метода применяют регулярное выражение к строке; оба указывают, с какой позиции начать и в какой закончить поиск по шаблону; оба возвращают объект `match` для первого найденного соответствия заданному шаблону. Разница между этими двумя методами состоит в том, что метод `match()` пытается отыскать совпадение только от начала строки или от указанного места в строке, не переходя в другие позиции в строке, а метод `search()` будет пытаться отыскать соответствие шаблону в любом месте строки или между начальной и конечной позицией, которые вы укажете, как показано в примере 3.20.

Пример 3.20. Сравнение методов match( ) и search( )

```
In [1]: import re

In [2]: re_obj = re.compile('FOO')

In [3]: search_string = ' FOO'

In [4]: re_obj.search(search_string)

Out[4]: <_sre.SRE_Match object at 0xa22f38>

In [5]: re_obj.match(search_string)

In [6]:
```

Даже при том, что в строке `search_string` имеется соответствие шаблону, поиск по которому производит метод `match()`, тем не менее, поиск завершается неудачей, потому что подстрока в `search_string`, соответствующая шаблону, находится не в начале строки. Метод `search()`, напротив, нашел соответствие и вернул объект `match`.

Методы `search()` и `match()` принимают параметры, определяющие начальную и конечную позицию поиска в строке, как показано в примере 3.21.

Пример 3.21. Параметры начала и конца поиска в методах search( ) и match( )

```
In [6]: re_obj.search(search_string, pos=1)

Out[6]: <_sre.SRE_Match object at 0xabe030>

In [7]: re_obj.match(search_string, pos=1)

Out[7]: <_sre.SRE_Match object at 0xabe098>
```

```
In [8]: re_obj.search(search_string, pos=1, endpos=3)

In [9]: re_obj.match(search_string, pos=1, endpos=3)

In [10]:
```

Параметр `pos` – это индекс, определяющий место в строке, откуда должен начинаться поиск по шаблону. В данном примере передача параметра `pos` методу `search()` не повлияла на результат, но передача параметра `pos` методу `match()`привела к тому, что он нашел соответствие шаблону, хотя без параметра `pos` соответствие обнаружить не удалось. Установка параметра `endpos` в значение 3 привела к тому, что оба метода – и `match()`, и `search()` не нашли соответствие, потому что соответствие шаблону включает символ в третьей позиции.

Методы `findall()` и `finditer()` отвечают на вопрос: «чему соответствует мой шаблон?», а главный вопрос, на который отвечают методы `search()` и `match()`: «имеется ли соответствие моему шаблону?». Методы `search()` и `match()` отвечают также на вопрос: «каково первое соответствие моему шаблону?», но часто единственное, что требуется узнать, это: «имеется ли соответствие моему шаблону?». Например, предположим, что необходимо написать сценарий, который должен читать строки из файла журнала и обертывать каждую строку в теги HTML, чтобы обеспечить удобочитаемое отображение. При этом хотелось бы, чтобы все строки, содержащие текст «ERROR», отображались красным цветом, для чего можно было бы выполнить цикл по всем строкам в файле, проверить их с помощью регулярного выражения и, если метод `search()` обнаруживает текст «ERROR», можно было бы определить такой формат строки, чтобы она отображалась красным цветом.

Методы `search()` и `match()` удобны не только тем, что они определяют наличие соответствия, но и тем, что они возвращают объект `match`. Объекты `match` содержат различные методы извлечения данных, которые могут пригодиться при обходе полученных результатов. Особый интерес представляют такие методы объекта `match`, как `start()`, `end()`, `span()`, `groups()` и `groupdict()`.

Методы `start()`, `end()` и `span()` определяют позиции в строке поиска, где совпадение с шаблоном начинается и где заканчивается. Метод `start()` возвращает целое число, определяющее позицию в строке начала найденного соответствия. Метод `end()` возвращает целое число, определяющее позицию в строке конца найденного соответствия. А метод `span()` возвращает кортеж, содержащий позицию начала и конца совпадения.

Метод `groups()` возвращает кортеж совпадения, каждый элемент которого соответствует группе, имеющейся в шаблоне. Этот кортеж напоминает кортежи в списке, возвращаемом методом `findall()`. Метод `groupdict()` возвращает словарь именованных групп, ключи которого соответствуют именам групп, присутствующих непосредственно в регулярном выражении, например: `(?P<group_name>pattern)`.

Подводя итоги, можно сказать – чтобы эффективно использовать регулярные выражения, следует взять в привычку использовать объекты скомпилированных регулярных выражений. Используйте методы `findall()` и `finditer()`, когда необходимо получить части текста, соответствующие шаблону. Запомните, что метод `finditer()` обладает более высокой гибкостью, чем `findall()`, потому что возвращает итератор по объектам `match`. Более подробный обзор библиотеки регулярных выражений вы найдете в главе 9 книги «Python in a Nutshell» Алекса Мартелли (Alex Martelli) (O'Reilly). Чтобы познакомиться с регулярными выражениями в действии, обращайтесь к книге «Data Crunching» Грега Уилсона (Greg Wilson) (The Pragmatic Bookshelf).

Работа с конфигурационным файлом Apache

Теперь, когда вы получили представление о работе с регулярными выражениями в языке Python, попробуем поработать с конфигурационным файлом веб-сервера Apache:

```
NameVirtualHost 127.0.0.1:80
<VirtualHost localhost:80>
    DocumentRoot /var/www/
    <Directory />
        Options FollowSymLinks
        AllowOverride None
    </Directory>
    ErrorLog /var/log/apache2/error.log
    LogLevel warn
    CustomLog /var/log/apache2/access.log combined
    ServerSignature On
</VirtualHost>
<VirtualHost local2:80>
    DocumentRoot /var/www2/
    <Directory />
        Options FollowSymLinks
        AllowOverride None
    </Directory>
    ErrorLog /var/log/apache2/error2.log
    LogLevel warn
    CustomLog /var/log/apache2/access2.log combined
    ServerSignature On
</VirtualHost>
```

Это слегка измененный конфигурационный файл Apache в Ubuntu. Мы создали именованные виртуальные хосты для некоторых своих нужд. Мы также добавили в файл */etc/hosts* следующую строку:

```
127.0.0.1     local2
```

Она позволяет указать броузеру, что серверу с именем `local2` соответствует IP-адрес 127.0.01, то есть локальный компьютер. И в чем же здесь смысл? Если в броузере ввести адрес *http://local2*, он передаст серверу

указанное имя в заголовке HTTP. Ниже приводится HTTP-запрос, направленный серверу `local2`:

```
GET / HTTP/1.1
Host: local2
User-Agent: Mozilla/5.0 (X11; U; Linux x86_64; en-US; rv:1.8.1.13)
Gecko/20080325 Ubuntu/7.10 (gutsy) Firefox/2.0.0.13
Accept: text/xml,application/xml,application/xhtml+xml,text/html
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Connection: keep-alive
If-Modified-Since: Tue, 15 Apr 2008 17:25:24 GMT
If-None-Match: "ac5ea-53-44aecaf804900"
Cache-Control: max-age=0
```

Обратите внимание, что запрос начинается с заголовка `Host:`. Когда веб-сервер Apache получит такой запрос, он направит его виртуальному хосту с именем `local2`.

Теперь все, что нам предстоит сделать, – это написать сценарий, который анализирует конфигурационный файл веб-сервера Apache, такой, как показано выше, отыскивает раздел `VirtualHost` и замещает значение параметра `DocumentRoot` в этом разделе. Сам сценарий приводится ниже:

```
#!/usr/bin/env python

from cStringIO import StringIO
import re

vhost_start = re.compile(r'<VirtualHost\s+(.*?)>')
vhost_end = re.compile(r'</VirtualHost')
docroot_re = re.compile(r'(DocumentRoot\s+)(\S+)')

def replace_docroot(conf_string, vhost, new_docroot):
    '''отыскивает в файле httpd.conf строки DocumentRoot, соответствующие
       указанному vhost, и замещает их новыми строками new_docroot
    '''
    conf_file = StringIO(conf_string)
    in_vhost = False
    curr_vhost = None
    for line in conf_file:
        vhost_start_match = vhost_start.search(line)
        if vhost_start_match:
            curr_vhost = vhost_start_match.groups()[0]
            in_vhost = True
        if in_vhost and (curr_vhost == vhost):
            docroot_match = docroot_re.search(line)
            if docroot_match:
                sub_line = docroot_re.sub(r'\1%s' % new_docroot, line)
                line = sub_line
```

```
        vhost_end_match = vhost_end.search(line)
        if vhost_end_match:
            in_vhost = False
        yield line

if __name__ == '__main__':
    import sys
    conf_file = sys.argv[1]
    vhost = sys.argv[2]
    docroot = sys.argv[3]
    conf_string = open(conf_file).read()
    for line in replace_docroot(conf_string, vhost, docroot):
        print line,
```

Этот сценарий сначала создает три объекта скомпилированных регулярных выражений: один соответствует открывающему тегу `VirtualHost`, один – закрывающему тегу `VirtualHost` и один – строке с параметром `DocumentRoot`. Мы также создали функцию, которая выполняет эту утомительную работу. Функция называется `replace_docroot` и принимает в качестве аргументов тело конфигурационного файла в виде строки, имя раздела `VirtualHost`, который требуется отыскать, и значение параметра `DocumentRoot`, которое требуется назначить для данного виртуального хоста. Функция устанавливает признак состояния, который указывает, находится ли текущая анализируемая строка в разделе `VirtualHost`. Кроме того, сохраняется имя текущего виртуального хоста. При анализе строк в разделе `VirtualHost` эта функция пытается отыскать строку с параметром `DocumentRoot` и изменяет его значение. Поскольку функция `replace_docroot()` выполняет итерации по каждой строке в конфигурационном файле, она возвращает либо неизмененную исходную строку, либо измененную строку с параметром `DocumentRoot`.

Мы создали простой интерфейс командной строки к этой функции. В нем не предусматривается использование ничего особенного, такого как функция `optparse`, и не выполняется проверка на количество входных аргументов, но он работает. Теперь попробуем применить этот сценарий к конфигурационному файлу веб-сервера Apache, представленному выше, и изменим настройки `VirtualHost local2:80` так, чтобы он использовал каталог */tmp* в качестве корневого каталога документов. Предусмотренный нами интерфейс командной строки просто выводит строки, возвращаемые функцией `replace_docroot()`, а не изменяет сам файл:

```
jmjones@dinkgutsy:code$ python apache_conf_docroot_replace.py
  /etc/apache2/sites-available/psa
  local2:80 /tmp
NameVirtualHost 127.0.0.1:80
<VirtualHost localhost:80>
        DocumentRoot /var/www/
        <Directory />
                Options FollowSymLinks
```

```
                AllowOverride None
        </Directory>
        ErrorLog /var/log/apache2/error.log
        LogLevel warn
        CustomLog /var/log/apache2/access.log combined
        ServerSignature On
</VirtualHost>
<VirtualHost local2:80>
        DocumentRoot /tmp
        <Directory />
                Options FollowSymLinks
                AllowOverride None
        </Directory>
        ErrorLog /var/log/apache2/error2.log
        LogLevel warn
        CustomLog /var/log/apache2/access2.log combined
        ServerSignature On
</VirtualHost>
```

Единственная строка, которая изменилась, – это строка с параметром `DocumentRoot` в разделе `VirtualHost local2:80`. Ниже приводятся различия, полученные после того, как вывод сценария был перенаправлен в файл:

```
jmjones@dinkgutsy:code$ diff apache_conf.diff /etc/apache2/sites-available/psa
20c20
< DocumentRoot /tmp
---
> DocumentRoot /var/www2/
```

Изменение значения параметра `DocumentRoot` в конфигурационном файле веб-сервера Apache – это достаточно простая задача, но когда это приходится делать достаточно часто или когда имеется множество виртуальных хостов, которые приходится изменять, тогда есть смысл написать сценарий, подобный тому, что был показан выше. Не менее просто можно было бы изменить сценарий так, чтобы он комментировал требуемый раздел `VirtualHost`, изменял значение параметра `LogLevel` или изменял имя файла журнала для указанного виртуального хоста.

Работа с файлами

Овладение приемами работы с файлами является ключом к обработке текстовых данных. Зачастую текст, который требуется обработать, находится в текстовом файле, например, в файле журнала, в конфигурационном файле или в файле с данными приложения. Нередко результаты анализа данных требуется сохранить в виде файла отчета или просто записать их в текстовый файл для последующего изучения. К счастью, в языке Python имеется простой в использовании тип объектов с именем `file`, который в состоянии помочь выполнить все необходимые действия с файлами.

Создание файлов

Это может показаться странным, но чтобы прочитать содержимое существующего файла, необходимо создать новый объект типа `file`. Однако не надо путать операцию создания нового объекта с созданием нового файла. Чтобы выполнить операцию записи в файл, необходимо создать новый объект `file` и, возможно, создать новый файл на диске, поэтому в такой ситуации создание объекта `file` интуитивно более понятно, чем создание объекта `file` для чтения. Создавать объект `file` обязательно, потому что он необходим для организации взаимодействий с файлом на диске.

Для создания объекта `file` используется встроенная функция `open()`. Ниже приводится фрагмент программного кода, который открывает файл для чтения:

```
In [1]: infile = open("foo.txt", "r")

In [2]: print infile.read()
Some Random
    Lines
Of
    Text.
```

Функция `open()` – это встроенная функция, поэтому нет никакой необходимости импортировать какой-либо модуль. Функция `open()` принимает три аргумента: имя файла, режим открытия и размер буфера. Обязательным является только первый аргумент – имя файла. Наиболее часто используются режимы: «r» (режим чтения, используется по умолчанию), «w» (режим записи) и «a» (режим записи в конец файла). Вместе с этими тремя спецификаторами режимов может использоваться дополнительный спецификатор «b», определяющий двоичный режим доступа. Третий аргумент, размер буфера, определяет способ буферизации операций над файлом.

В предыдущем примере было предписано открыть файл *foo.txt* в режиме для чтения и сохранить ссылку на созданный объект файла в переменной `infile`. После получения ссылки на объект в переменной `infile` появилась возможность обратиться к методу `read()` этого объекта, который читает содержимое файла целиком.

Создание объекта типа `file` для записи в файл выполняется почти так же, как создание объекта для чтения из файла. Просто вместо спецификатора режима `"r"` следует использовать спецификатор `"w"`:

```
In [1]: outputfile = open("foo_out.txt", "w")

In [2]: outputfile.write("This is\nSome\nRandom\nOutput Text\n")

In [3]: outputfile.close()
```

В этом примере предписывается открыть файл *foo_out.txt* в режиме для записи и сохранить ссылку на вновь созданный объект типа `file`

в переменной outputfile. После получения ссылки на объект мы смогли обратиться к методу write(), чтобы записать в файл некоторый текст и закрыть его вызовом метода close().

Несмотря на всю простоту создания файлов, у вас может появиться желание создавать файлы способом, более устойчивым к появлению ошибок. Считается хорошей практикой обертывать вызов функции open() конструкцией try/finally, особенно, когда вслед за этим вызывается метод write(). Ниже приводится пример реализации записи в файл с использованием инструкции try/finally:

```
In [1]: try:
   ...:     f = open('writeable.txt', 'w')
   ...:     f.write('quick line here\n')
   ...: finally:
   ...:     f.close()
```

При такой реализации записи файлов метод close() вызывается, когда где-нибудь в блоке try возникает исключение. В действительности этот подход позволяет методу close() закрыть файл, даже когда в блоке try не возникает исключение. Блок finally выполняется после завершения работы блока try всегда, независимо от того, возникло исключение или нет.

В версии Python 2.5 появилась новая идиома – инструкция with, которая позволяет использовать менеджер контекста. Менеджер контекста – это просто объект с методами __enter__() и __exit__(). Когда объект создается с помощью инструкции with, вызывается метод __enter__() менеджера контекста. Когда выполнение блока with завершается, вызывается метод __exit__() менеджера контекста, даже если возникло исключение. Объекты типа file имеют методы __enter__() и __exit__(). В методе __exit__() объекта типа file вызывается метод close(). Ниже приводится пример использования инструкции with:

```
In [1]: from __future__ import with_statement

In [2]: with open('writeable.txt', 'w') as f:
   ...:     f.write('this is a writeable file\n')
   ...:
   ...:
```

Хотя в этом фрагменте отсутствует вызов метода close() объекта f, менеджер контекста закроет файл после выхода из блока with:

```
In [3]: f
Out[3]: <closed file 'writeable.txt', mode 'w' at 0x1382770>

In [4]: f.write("this won't work")
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)

/Users/jmjones/<ipython console> in <module>()
```

```
ValueError: I/O operation on closed file
(ValueError: операция ввода-вывода с закрытым файлом)
```

Как и следовало ожидать, файл был закрыт. Хотя это хорошая практика – обрабатывать все возможные исключения и гарантировать закрытие файла, когда это необходимо, но ради простоты и ясности мы не будем предусматривать такую обработку во всех примерах.

Полный перечень методов объектов типа `file` вы найдете в разделе «File Objects» в справочнике «Python Library Reference» по адресу: *http://docs.python.org/lib/bltin-file-objects.html*.

Чтение из файлов

Как только появляется объект файла, открытого для чтения с флагом `r`, вы получаете возможность использовать три обычных метода объекта `file`, удобных для получения данных, содержащихся в файле: `read()`, `readline()` и `readlines()`. Метод `read()` читает, что не удивительно, данные из объекта открытого файла и возвращает эти данные в виде строки. Метод `read()` принимает необязательный аргумент, который указывает число байтов, которые требуется прочитать из файла. Если этот аргумент отсутствует, метод `read()` попытается прочитать содержимое файла целиком. Если размер файла меньше, чем величина аргумента, метод `read()` будет читать данные, пока не встретит конец файла и вернет то, что удалось прочитать.

Допустим, что имеется следующий файл:

```
jmjones@dink:~/some_random_directory$ cat foo.txt Some Random
    Lines
Of
    Text.
```

Тогда метод `read()` будет работать с этим файлом, как показано ниже:

```
In [1]: f = open("foo.txt", "r")
In [2]: f.read()
Out[2]: 'Some Random\n    Lines\nOf \n    Text.\n'
```

Обратите внимание, что символы новой строки отображаются как последовательности `\n` – это стандартный способ обозначения символа новой строки.

Если бы требовалось прочитать только первые 5 байтов, сделать это можно было бы следующим способом:

```
In [1]: f = open("foo.txt", "r")

In [2]: f.read(5)
Out[2]: 'Some '
```

Следующий метод, позволяющий получать текст из файла, – метод `readline()`. Метод `readline()` читает текст из файла по одной строке за

раз. Этот метод принимает один необязательный аргумент: `size`. Он определяет максимальное число байтов, которые метод `readline()` будет читать из файла, прежде чем вернуть строку, независимо от того, был достигнут конец строки или нет. Поэтому в следующем примере программа читает первую строку из текста из файла *foo.txt*, затем читает первые 7 байтов текста из второй строки, а после этого считывает остаток второй строки:

```
In [1]: f = open("foo.txt", "r")

In [2]: f.readline()
Out[2]: 'Some Random\n'

In [3]: f.readline(7)
Out[3]: ' Lin'

In [4]: f.readline()
Out[4]: 'es\n'
```

Последний метод получения текста из объектов типа `file`, который мы рассмотрим, – это метод `readlines()`. Имя `readlines()` – это не опечатка и не ошибка, закравшаяся при копировании имени метода из предыдущего примера. Метод `readlines()` читает сразу все строки из файла. Впрочем, это почти правда. Метод `readlines()` имеет аргумент `sizehint`, определяющий максимальное число байтов, которые требуется прочитать. В следующем примере мы создали файл *biglines.txt*, содержащий 10 000 строк, в каждой из которых по 80 символов. После этого мы открыли файл, указали, что нам требуется прочитать из файла N первых строк, общий объем которых составляет примерно 1024 байта, определили число прочитанных строк и байтов и затем прочитали оставшуюся часть файла:

```
In [1]: f = open("biglines.txt", "r")

In [2]: lines = f.readlines(1024)

In [3]: len(lines)
Out[3]: 102

In [4]: len("".join(lines))
Out[4]: 8262

In [5]: lines = f.readlines()

In [6]: len(lines)
Out[6]: 9898

In [7]: len("".join(lines))
Out[7]: 801738
```

Команда в строке [3] показывает, что было прочитано 102 строки, а команда в строке [4] показала, что общее число прочитанных байтов составило 8262. Как так вышло, что мы указали «примерное» число байтов, которые требуется прочитать, равное 1024, а получили 8262? Оно

было округлено до размера внутреннего буфера, который равен примерно 8 килобайтам. Суть в том, что аргумент `sizehint` не всегда оказывает влияние так, как вам того хотелось бы, и об этом следует помнить.

Запись в файлы

Иногда возникает потребность не только читать данные из файлов, но также создавать собственные файлы и записывать в них данные. Объекты типа `file` обладают двумя основными методами, которые позволят вам записывать данные в файлы. Первый метод, который уже демонстрировался выше, – это метод `write()`. Метод `write()` принимает один аргумент: строку, которую требуется записать в файл. В следующем примере демонстрируется запись данных в файл:

```
In [1]: f = open("some_writable_file.txt", "w")

In [2]: f.write("Test\nFile\n")

In [3]: f.close()

In [4]: g = open("some_writable_file.txt", "r")

In [5]: g.read()
Out[5]: 'Test\nFile\n'
```

Команда [1] открывает файл с флагом режима `w`, то есть в режиме для записи. Команда [2] записывает в файл две строки. Команда [4] создает новый объект файла и присваивает ссылку на него другой переменной, с именем `g`, чтобы избежать путаницы, хотя вполне возможно было использовать и переменную `f`. И команда [5] показывает, что метод `read()` возвращает те же самые данные, которые были записаны в файл.

Следующий основной метод записи данных в файл – это метод `writelines()`. Метод `writelines()` принимает один обязательный параметр: последовательность, которая должна быть записана в файл. Допускается использовать последовательности любого итерируемого типа, такие как списки, кортежи, генераторы списков (которые можно считать списками) или генераторы. Ниже приводится пример вызова метода `writelines()`, который получает данные для записи в файл от выражения-генератора:

```
In [1]: f = open("writelines_outfile.txt", "w")

In [2]: f.writelines("%s\n" % i for i in range(10))

In [3]: f.close()

In [4]: g = open("writelines_outfile.txt", "r")

In [5]: g.read()

Out[5]: '0\n1\n2\n3\n4\n5\n6\n7\n8\n9\n'
```

И еще один пример функции-генератора, которая может использоваться для записи данных в файл (этот пример функционально эквива-

лентен предыдущему, но для его реализации потребовалось написать больше программного кода):

```
In [1]: def myRange(r):
   ...:     i = 0
   ...:     while i < r:
   ...:         yield "%s\n" % i
   ...:         i += 1
   ...:
   ...:

In [2]: f = open("writelines_generator_function_outfile", "w")

In [3]: f.writelines(myRange(10))

In [4]: f.close()

In [5]: g = open("writelines_generator_function_outfile", "r")

In [6]: g.read()
Out[6]: '0\n1\n2\n3\n4\n5\n6\n7\n8\n9\n'
```

Следует заметить, что метод `writelines()` не записывает символы новой строки (`\n`) автоматически – вы сами должны включать их в последовательность, предназначенную для записи в файл. Кроме того, следует знать, что этот метод можно использовать не только для построчной записи данных в файл. Возможно, этому методу лучше подошло бы название `writeiter()`.Так случилось, что в предыдущих примерах мы записывали текст, который уже содержал символы новой строки, но нет никаких причин, которые требовали бы их наличия.

Дополнительные ресурсы

За дополнительной информацией об объектах типа `file` обращайтесь к главе 7 в книге «Learning Python» Дэвида Ашера (David Ascher) и Марка Лутца (Mark Lutz) (O'Reilly) (имеется также в Интернете по адресу: *http://safari.oreilly.com/0596002815/lpython2-chp-7-sect-2*) или к разделу «File Objects» в справочнике «Python Library Reference» (доступному в Интернете по адресу: *http://docs.python.org/lib/bltin-file-objects.html*).

За дополнительной информацией о выражениях-генераторах обращайтесь к разделу «generator expressions» в справочнике «Python Language Reference» (доступному в Интернете по адресу: *http://docs.python.org/ref/genexpr.html*). За дополнительной информацией об инструкции `yield` обращайтесь к разделу «yield statement» в справочнике «Python Language Reference» (доступному в Интернете по адресу: *http://docs.python.org/ref/yield.html*).

Стандартный ввод и вывод

Операции чтения текста из потока стандартного ввода процесса и записи в поток стандартного вывода процесса знакомы большинству сис-

темных администраторов. Стандартный ввод – это обычные данные, поступающие в программу, которые программа может читать в ходе своей работы. Стандартный вывод – это данные, которые программа выводит в процессе выполнения. Преимущество использования стандартного ввода и вывода состоит в том, что это позволяет объединять команды в конвейеры с другими утилитами.

Стандартная библиотека языка Python содержит встроенный модуль с именем `sys`, который обеспечивает простые способы доступа к стандартному вводу и стандартному выводу. Стандартная библиотека предоставляет доступ к стандартному вводу и выводу как к объектам типа `file`, несмотря на то, что они не имеют никакого отношения к файлам на диске. А так как эти объекты напоминают объекты типа `file`, для работы с ними можно использовать те же самые методы, которые используются при работе с файлами. Вы можете работать с ними, как если бы это были файлы на диске, и обращаться к соответствующим методам для выполнения требуемых операций.

После импортирования модуля `sys` стандартный ввод становится доступен в виде атрибута `stdin` этого модуля (`sys.stdin`). Атрибут `sys.stdin` – это доступный для чтения объект типа `file`. Обратите внимание, что произойдет, если создать «настоящий» объект типа `file`, открыв файл с именем *foo.txt* на диске, и затем сравнить его с объектом `sys.stdin`:

```
In [1]: import sys

In [2]: f = open("foo.txt", "r")

In [3]: sys.stdin

Out[3]: <open file '<stdin>', mode 'r' at 0x14020>

In [4]: f
Out[4]: <open file 'foo.txt', mode 'r' at 0x12179b0>

In [5]: type(sys.stdin) == type(f)
Out[5]: True
```

Интерпретатор воспринимает их как объекты одного и того же типа, поэтому они обладают одними и теми же методами. Несмотря на то, что с технической точки зрения эти объекты принадлежат одному и тому же типу и обладают одними и теми же методами, поведение некоторые методов будет отличаться. Например, методы `sys.stdin.seek()` и `sys.stdin.tell()` доступны, но при обращении к ним возбуждается исключение (в данном случае исключение `IOError`). Однако во всем остальном стандартный ввод и вывод напоминают объекты типа `file`, и вы в значительной степени можете воспринимать их как обычные дисковые файлы.

Доступ к `sys.stdin` в оболочке Python (или в оболочке IPython) практически лишен всякого смысла. Попытка импортировать модуль `sys` и вызвать метод `sys.stdin.read()` просто заблокирует работу оболочки. Чтобы продемонстрировать вам, как работает объект `sys.stdin`, мы

написали сценарий, который читает данные из sys.stdin и выводит обратно каждую прочитанную строку с соответствующим ей номером, как показано в примере 3.22.

Пример 3.22. Нумерация строк, читаемых методом sys.stdin.readline

```
#!/usr/bin/env python

import sys

counter = 1
while True:
    line = sys.stdin.readline()
    if not line:
        break
    print "%s: %s" % (counter, line)
    counter += 1
```

В этом примере мы создали переменную counter, с помощью которой сценарий следит за номерами введенных строк. Далее следует цикл while, в теле которого выполняется чтение строк со стандартного ввода. Для каждой строки вводится ее порядковый номер и ее содержимое. Так как программа все время находится в процессе выполнения цикла, она обрабатывает все строки, которые ей поступают, даже если они оказываются пустыми. Но даже пустые строки – не совсем пустые: они содержат символ новой строки (\n). Когда сценарий обнаруживает признак «конца файла», он прерывает работу цикла.

Ниже приводится результат объединения в конвейер команды who и предыдущего сценария:

```
jmjones@dink:~/psabook/code$ who | ./sys_stdin_readline.py
1: jmjones console Jul 9 11:01

2: jmjones ttyp1 Jul 9 19:58

3: jmjones ttyp2 Jul 10 05:10

4: jmjones ttyp3 Jul 11 11:51

5: jmjones ttyp4 Jul 13 06:48

6: jmjones ttyp5 Jul 11 21:49

7: jmjones ttyp6 Jul 15 04:38
```

Достаточно интересно, что предыдущий пример можно реализовать гораздо проще и короче, если использовать функцию enumerate(), как показано в примере 3.23.

Пример 3.23. Пример использования метода sys.stdin.readline( )

```
#!/usr/bin/env python

import sys

for i, line in enumerate(sys.stdin):
    print "%s: %s" % (i, line)
```

Чтобы получить доступ к стандартному вводу, необходимо импортировать модуль `sys` и затем воспользоваться атрибутом `stdin`. Точно так же, чтобы получить доступ к стандартному выводу, необходимо импортировать модуль `sys` и воспользоваться атрибутом `stdout`. Так же, как `sys.stdin` представляет объект файла, доступного для чтения, объект `sys.stdout` представляет объект файла, доступного для записи. И так же, как `sys.stdin` имеет тот же тип, что и объект файла, доступного для чтения, объект `sys.stdout` имеет тот же тип, что и объект файла, доступного для записи:

```
In [1]: import sys

In [2]: f = open('foo.txt', 'w')

In [3]: sys.stdout
Out[3]: <open file '<stdout>', mode 'w' at 0x14068>

In [4]: f
Out[4]: <open file 'foo.txt', mode 'w' at 0x1217968>

In [5]: type(sys.stdout) == type(f)
Out[5]: True
```

Важное замечание: следующее утверждение не должно быть неожиданным, поскольку любой файл, открытый для чтения, и любой файл, открытый для записи, относятся к одному и тому же типу:

```
In [1]: readable_file = open('foo.txt', 'r')

In [2]: writable_file = open('foo_writable.txt', 'w')

In [3]: readable_file

Out[3]: <open file 'foo.txt', mode 'r' at 0x1243530>

In [4]: writable_file
Out[4]: <open file 'foo_writable.txt', mode 'w' at 0x1217968>

In [5]: type(readable_file) == type(writable_file)
Out[5]: True
```

Важно знать, что `sys.stdout` может в значительной степени рассматриваться как объект типа `file`, открытый для записи, точно так же как и `sys.stdin` может рассматриваться как объект типа `file`, открытый для чтения.

StringIO

Как быть в случае, когда функция, выполняющая обработку текста, предназначена для работы с объектом типа `file`, а данные, которые предстоит обрабатывать, доступны в виде текстовой строки, а не в виде объекта `file`? Самое простое решение состоит в том, чтобы воспользоваться модулем `StringIO`:

```
In [1]: from StringIO import StringIO

In [2]: file_like_string = StringIO("This is a\nmultiline string.\n
```

```
readline() should see\nmultiple lines of\ninput")

In [3]: file_like_string.readline()
Out[3]: 'This is a\n'

In [4]: file_like_string.readline()
Out[4]: 'multiline string.\n'

In [5]: file_like_string.readline()
Out[5]: 'readline() should see\n'

In [6]: file_like_string.readline()
Out[6]: 'multiple lines of\n'

In [7]: file_like_string.readline()
Out[7]: 'input'
```

В этом примере мы создали объект `StringIO`, конструктору которого передается строка `"This is a\nmultiline string. \nreadline() should see\nmultiple lines of\ninput"`. После этого появилась возможность вызывать метод `readline()` объекта `StringIO`. Хотя в этом примере использовался только метод `readline()`, это далеко не единственный доступный метод, заимствованный у объектов типа `file`:

```
In [8]: dir(file_like_string)
Out[8]:
['__doc__',
'__init__',
'__iter__',
'__module__',
'buf',
'buflist',
'close',
'closed',
'flush',
'getvalue',
'isatty',
'len',
'next',
'pos',
'read',
'readline',
'readlines',
'seek',
'softspace',
'tell',
'truncate',
'write',
'writelines']
```

Конечно, между файлами и строками существуют различия, но интерфейс позволяет легко переходить между использованием файлов и строк. Ниже приводится сравнение методов и атрибутов объектов типа `file` и объектов типа `StringIO`:

```
In [9]: f = open("foo.txt", "r")

In [10]: from sets import Set

In [11]: sio_set = Set(dir(file_like_string))

In [12]: file_set = Set(dir(f))

In [13]: sio_set.difference(file_set)
Out[13]: Set(['__module__', 'buflist', 'pos', 'len', 'getvalue', 'buf'])

In [14]: file_set.difference(sio_set)
Out[14]: Set(['fileno', '__setattr__', '__reduce_ex__', '__new__',
'encoding',
'__getattribute__', '__str__', '__reduce__', '__class__', 'name',
'__delattr__', 'mode', '__repr__', 'xreadlines', '__hash__', 'readinto',
'newlines'])
```

Как видите, если возникнет необходимость работать со строкой как с файлом, объект типа `StringIO` может оказать существенную помощь.

urllib

Что, если интересующий вас файл находится где-то в сети? Или вы хотите использовать уже существующий фрагмент программного кода, который предполагает работу с объектом `file`? Встроенный тип `file` не умеет взаимодействовать с сетью, и в этой ситуации вам поможет модуль `urllib`.

Если вам требуется только вызвать метод `read()` для получения данных файла, расположенного на некотором веб-сервере, для этого можно просто воспользоваться методом `urllib.urlopen()`, как показано в простом примере ниже:

```
In [1]: import urllib

In [2]: url_file = urllib.urlopen("http://docs.python.org/lib/module-
urllib.html")

In [3]: urllib_docs = url_file.read()

In [4]: url_file.close()

In [5]: len(urllib_docs)
Out[5]: 28486

In [6]: urllib_docs[:80]
Out[6]: '<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">\
  n<html>\n<head>\n<li'

In [7]: urllib_docs[-80:]
Out[7]: 'nt...</a></i> for information on suggesting changes.\
  n</address>\n</body>\n</html>\n'
```

Здесь сначала импортируется модуль `urllib`. Затем создается `file`-подобный объект с именем `url_file`. Далее выполняется чтение содержимого `url_file` в строку с именем `urllib_docs`. И только для того, чтобы

продемонстрировать, что данные действительно были получены из Интернета, с помощью операции извлечения среза выводятся первые и последние 80 символов полученного документа. Обратите внимание, что объекты файлов, созданные средствами `urllib`, поддерживают методы `read()` и `close()`. Кроме того, они поддерживают методы `readline()`, `readlines()`, `fileno()`, `info()` и `geturl()`.

Если вам потребуются более широкие возможности, такие как работа через прокси-сервер, ищите дополнительную информацию о модуле `urllib` по адресу: *http://docs.python.org/lib/module-urllib.html*. Если вам требуются еще более широкие возможности, такие как аутентификация и работа с cookie, подумайте об использовании модуля `urllib2`, описание которого вы найдете по адресу: *http://docs.python.org/lib/module-urllib2.html*.

Анализ журналов

С точки зрения системного администратора никакое обсуждение вопросов обработки текста не может считаться законченным без обсуждения проблемы анализа файлов журналов, поэтому здесь мы рассмотрим эту проблему. Мы заложили основу знаний, которые позволят вам открыть файл журнала, прочитать его построчно и при этом извлекать данные тем способом, который вы сочтете наиболее подходящим. Прежде чем приступить к реализации очередного примера, нам необходимо ответить на вопрос: «Что нам необходимо получить в результате чтения файла журнала?». Ответ достаточно прост: прочитать журнал обращений к веб-серверу Apache и определить количество байтов, полученных каждым отдельным клиентом.

Согласно описанию *http://httpd.apache.org/docs/1.3/logs.html* «комбинированный» формат записи в файле журнала имеет следующий вид:

```
127.0.0.1 - frank [10/Oct/2000:13:55:36 -0700] "GET /apache_pb.gif HTTP/1.0"
200 2326 "http://www.example.com/start.html" "Mozilla/4.08 [en] (Win98; I
;Nav)"
```

И это соответствует данным в нашем файле журнала веб-сервера Apache. В каждой строке журнала интерес для нас будут представлять две вещи: IP-адрес клиента и число переданных байтов. IP-адрес клиента находится в первом поле записи, в данном случае – это адрес 127.0.0.1. Количество переданных байтов содержится в третьем поле с конца, в данном случае было передано 2326 байтов. Как же нам получить эти поля? Взгляните на пример 3.24.

Пример 3.24. Сценарий анализа файла журнала веб-сервера Apache – разбиение по пробелам

```
#!/usr/bin/env python
"""
```

```
ПОРЯДОК ИСПОЛЬЗОВАНИЯ:

apache_log_parser_split.py some_log_file

Этот сценарий принимает единственный аргумент командной строки: имя файла
журнала, который требуется проанализировать. Он анализирует содержимое файла
и генерирует отчет, содержащий перечень удаленных хостов и число байтов,
переданных каждому из них.
"""

import sys

def dictify_logline(line):
    '''возвращает словарь, содержащий информацию, извлеченную из
    комбинированного файла журнала

    В настоящее время нас интересуют только адреса удаленных хостов
    и количество переданных байтов, но для полноты картины мы
    добавили выборку кода состояния.
    '''
    split_line = line.split()
    return {'remote_host': split_line[0],
            'status': split_line[8],
            'bytes_sent': split_line[9],
    }

def generate_log_report(logfile):
    '''возвращает словарь в формате:
       remote_host=>[список числа переданных байтов]

    Эта функция принимает объект типа file, выполняет обход всех строк
    в файле и создает отчет о количестве байтов, переданных при каждом
    обращении удаленного хоста к веб-серверу.
    '''
    report_dict = {}
    for line in logfile:
        line_dict = dictify_logline(line)
        print line_dict
        try:
            bytes_sent = int(line_dict['bytes_sent'])
        except ValueError:
            ##полностью игнорировать непонятные нам ошибки
            continue
        report_dict.setdefault(line_dict['remote_host'],
                               []).append(bytes_sent)
    return report_dict

if __name__ == "__main__":
    if not len(sys.argv) > 1:
        print __doc__
        sys.exit(1)
    infile_name = sys.argv[1]
    try:
        infile = open(infile_name, 'r')
```

```
    except IOError:
        print "You must specify a valid file to parse"
        print __doc__
        sys.exit(1)
    log_report = generate_log_report(infile)
    print log_report
    infile.close()
```

Этот пример чрезвычайно прост. В разделе `__main__` выполняется всего несколько действий. Во-первых, осуществляется минимально необходимая проверка аргументов командной строки, чтобы убедиться, что сценарий получил как минимум один аргумент. Если пользователь запустит сценарий без аргументов, сценарий выведет сообщение о порядке использования и завершит работу. Более полное обсуждение, как лучше обрабатывать аргументы и параметры командной строки, приводится в главе 13. Далее, в разделе `__main__` предпринимается попытка открыть указанный файл журнала. Если попытка открыть файл завершается неудачей, сценарий выведет сообщение о порядке использования и завершит работу. После этого сценарий передает файл функции `generate_log_report()` и выводит результаты.

Функция `generate_log_report()` создает словарь, который играет роль отчета. После этого она выполняет обход всех строк в файле и передает каждую строку функции `dictify_logline()`, которая в свою очередь возвращает словарь, содержащий необходимую нам информацию. Затем она проверяет, является ли значение `bytes_sent` целым числом. Если это целое число, обработка строки продолжается, если нет – выполняется переход к следующей строке. После этого она добавляет в словарь отчета данные, полученные от функции `dictify_logline()`. Наконец, она возвращает сформированный словарь отчета программному коду в разделе `__main__`.

Функция `dictify_logline()` просто разбивает строку по пробелам, извлекает определенные элементы из полученного списка и возвращает словарь с данными, извлеченными из строки.

Будет ли работать такой сценарий? Проверим это с помощью модульного теста из примера 3.25.

Пример 3.25. Модульный тест сценария анализа файла журнала веб-сервера Apache – разбиение по пробелам

```
#!/usr/bin/env python

import unittest
import apache_log_parser_split

class TestApacheLogParser(unittest.TestCase):

    def setUp(self):
        pass

    def testCombinedExample(self):
```

```
            # тест комбинированного примера с сайта apache.org
            combined_log_entry = '127.0.0.1 \
            '- frank [10/Oct/2000:13:55:36 -0700] '\
            '"GET /apache_pb.gif HTTP/1.0" 200 2326 '\
            '"http://www.example.com/start.html" '\
            '"Mozilla/4.08 [en] (Win98; I ;Nav)"'
            self.assertEqual(
                apache_log_parser_split.dictify_logline(combined_log_entry),
                {'remote_host':'127.0.0.1', 'status':'200', 'bytes_sent':'2326'})

        def testCommonExample(self):
            # тест общего примера с сайта apache.org
            common_log_entry = '127.0.0.1 '\
            '- frank [10/Oct/2000:13:55:36 -0700] '\
            '"GET /apache_pb.gif HTTP/1.0" 200 2326'
            self.assertEqual(
                apache_log_parser_split.dictify_logline(common_log_entry),
                {'remote_host':'127.0.0.1', 'status':'200', 'bytes_sent':'2326'})

        def testExtraWhitespace(self):
            # тест для случая с дополнительными пробелами между полями
            common_log_entry = '127.0.0.1      '\
            '-     frank [10/Oct/2000:13:55:36 -0700] '\
            '"GET /apache_pb.gif HTTP/1.0" 200 2326'
            self.assertEqual(
                apache_log_parser_split.dictify_logline(common_log_entry),
                {'remote_host':'127.0.0.1', 'status':'200', 'bytes_sent':'2326'})

        def testMalformed(self):
            # тест для случая с дополнительными пробелами между полями
            common_log_entry = '127.0.0.1      '\
            '-     frank [10/Oct/2000:13:55:36 -0700] '\
            '"GET /some/url/with white space.html HTTP/1.0" 200 2326'
            self.assertEqual(
                apache_log_parser_split.dictify_logline(common_log_entry),
                {'remote_host':'127.0.0.1', 'status':'200', 'bytes_sent':'2326'})

    if __name__ == '__main__':
        unittest.main()
```

Этот сценарий работает с комбинированным и общим форматами записей в журнале, но небольшое изменение в строке приводит к тому, что тест завершается неудачей. Ниже приводятся результаты тестирования:

```
jmjones@dinkgutsy:code$ python test_apache_log_parser_split.py
...F
======================================================================
FAIL: testMalformed (__main__.TestApacheLogParser)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "test_apache_log_parser_split.py", line 38, in testMalformed
    {'remote_host': '127.0.0.1', 'status': '200', 'bytes_sent': '2326'})
```

```
AssertionError: {'status': 'space.html', 'bytes_sent': 'HTTP/1.0"',
'remote_host': '127.0.0.1'} != {'status': '200', 'bytes_sent': '2326',
'remote_host': '127.0.0.1'}

----------------------------------------------------------------------
Ran 4 tests in 0.001s

FAILED (failures=1)
```

Вследствие того, что в поле адреса появились два лишних пробела, все последующие поля в этой записи сместились на две позиции вправо. Здоровая подозрительность – хорошее качество. Основываясь на спецификациях форматов записей в журнале, можно достаточно уверенно извлекать адреса удаленных хостов и число переданных байтов, опираясь на способ выделения полей по пробелам. Пример 3.26 представляет реализацию того же самого сценария, выполненную с применением регулярных выражений.

Пример 3.26. Сценарий анализа файла журнала веб-сервера Apache

```
#!/usr/bin/env python
"""
ПОРЯДОК ИСПОЛЬЗОВАНИЯ:

apache_log_parser_regex.py some_log_file

Этот сценарий принимает единственный аргумент командной строки: имя файла
журнала, который требуется проанализировать. Он анализирует содержимое файла
и генерирует отчет, содержащий перечень удаленных хостов и число байтов,
переданных каждому из них.
"""
import sys
import re
log_line_re = re.compile(r'''(?P<remote_host>\S+) #IP ADDRESS
                             \s+ #whitespace
                             \S+ #remote logname
                             \s+ #whitespace
                             \S+ #remote user
                             \s+ #whitespace
                             \[[^\[\]]+\] #time
                             \s+ #whitespace
                             "[^"]+" #first line of request
                             \s+ #whitespace
                             (?P<status>\d+)
                             \s+ #whitespace
                             (?P<bytes_sent>-|\d+)
                             \s* #whitespace
                             ''', re.VERBOSE)

def dictify_logline(line):
    '''возвращает словарь, содержащий информацию, извлеченную
    из комбинированного файла журнала
```

```
    В настоящее время нас интересуют только адреса удаленных хостов
    и количество переданных байтов, но но для полноты картины
    мы добавили выборку кода состояния.
    '''
    m = log_line_re.match(line)
    if m:
        groupdict = m.groupdict()
        if groupdict['bytes_sent'] == '-':
            groupdict['bytes_sent'] = '0'
        return groupdict
    else:
        return {'remote_host': None,
                'status': None,
                'bytes_sent': "0",
        }

def generate_log_report(logfile):
    '''возвращает словарь в формате:
       remote_host=>[список числа переданных байтов]

    Эта функция принимает объект типа file, выполняет обход
    всех строк в файле и создает отчет о количестве байтов,
    переданных при каждом обращении удаленного хоста к веб-серверу.
    '''
    report_dict = {}
    for line in logfile:
        line_dict = dictify_logline(line)
        print line_dict
        try:
            bytes_sent = int(line_dict['bytes_sent'])
        except ValueError:
            ##полностью игнорировать непонятные нам ошибки
            continue
        report_dict.setdefault(line_dict['remote_host'],
                               []).append(bytes_sent)
    return report_dict

if __name__ == "__main__":
    if not len(sys.argv) > 1:
        print __doc__
        sys.exit(1)
    infile_name = sys.argv[1]
    try:
        infile = open(infile_name, 'r')
    except IOError:
        print "You must specify a valid file to parse"
        print __doc__
        sys.exit(1)
    log_report = generate_log_report(infile)
    print log_report
    infile.close()
```

Единственная функция, которая изменилась по сравнению с примером, основанным на «разбиении по пробелам», – это функция `dictify_logline()`. При этом подразумевается, что тип значения, возвращаемого этой функцией, остался прежним и в примере, основанном на применении регулярных выражений. Вместо того, чтобы разбивать строку из журнала по пробелам, мы воспользовались объектом скомпилированного регулярного выражения, `log_line_re`, для выявления соответствий с помощью метода `match()`. Если соответствие обнаружено, с помощью метода `groupdict()` извлекаются практически готовые к возврату данные, где значение `bytes_sent` устанавливается равным 0, если поле содержит прочерк (-) (потому что прочерк означает «нисколько»). Если соответствие не было найдено, возвращается словарь с теми же самыми ключами, но со значениями элементов, равными `None` и 0.

Действительно ли версия сценария, основанная на использовании регулярных выражений, работает лучше, чем предыдущая? Да, это так. Ниже приводится модульный тест для новой версии сценария анализа файлов журнала веб-сервера Apache:

```
#!/usr/bin/env python

import unittest
import apache_log_parser_regex

class TestApacheLogParser(unittest.TestCase):

    def setUp(self):
        pass

    def testCombinedExample(self):
        # тест комбинированного примера с сайта apache.org
        combined_log_entry = '127.0.0.1 \
        '- frank [10/Oct/2000:13:55:36 -0700] '\
        '"GET /apache_pb.gif HTTP/1.0" 200 2326 '\
        '"http://www.example.com/start.html" '\
        '"Mozilla/4.08 [en] (Win98; I ;Nav)"'
        self.assertEqual(
            apache_log_parser_regex.dictify_logline(combined_log_entry),
            {'remote_host':'127.0.0.1', 'status':'200', 'bytes_sent':'2326'})

    def testCommonExample(self):
        # тест общего примера с сайта apache.org
        common_log_entry = '127.0.0.1 '\
        '- frank [10/Oct/2000:13:55:36 -0700] '\
        '"GET /apache_pb.gif HTTP/1.0" 200 2326'
        self.assertEqual(
            apache_log_parser_regex.dictify_logline(common_log_entry),
            {'remote_host':'127.0.0.1', 'status':'200', 'bytes_sent':'2326'})


    def testMalformed(self):
        # тест для модифицированного примера с ошибками с сайта apache.org
        #malformed_log_entry = '127.0.0.1 '\
```

```
            #'- frank [10/Oct/2000 13:55:36 -0700] '\
            #'"GET /apache_pb.gif HTTP/1.0" 200 2326 '\
            #'"http://www.example.com/start.html" '\
            #'"Mozilla/4.08 [en] (Win98; I ;Nav)"'

            malformed_log_entry = '127.0.0.1 '\
            '- frank [10/Oct/2000:13:55:36 -0700] '\
            '"GET /some/url/with white space.html HTTP/1.0" 200 2326'
            self.assertEqual(
                apache_log_parser_regex.dictify_logline(common_log_entry),
                {'remote_host':'127.0.0.1', 'status':'200', 'bytes_sent':'2326'})

if __name__ == '__main__':
    unittest.main()
```

И ниже – результаты модульного тестирования:

```
jmjones@dinkgutsy:code$ python test_apache_log_parser_regex.py
...
----------------------------------------------------------------------
Ran 3 tests in 0.001s

OK
```

ElementTree

Если текст, который необходимо проанализировать, имеет формат XML, скорее всего вам придется подходить к решению этой проблемы с несколько иной стороны, чем, например, к анализу обычных текстовых файлов журналов. Едва ли вы захотите читать такие файлы строку за строкой и выполнять поиск по шаблону, и едва ли получится широко использовать регулярные выражения. В формате XML используется древовидная структура организации данных, поэтому подход, основанный на построчном чтении, здесь не годится. И использование регулярных выражений для построения древовидной структуры данных легко может превратиться в кошмар.

Что же тогда делать? Для работы с форматом XML обычно используется один из двух подходов. Существует такая вещь, как «simple API for XML» (простой прикладной интерфейс для работы с форматом XML), или SAX. Стандартная библиотека языка Python имеет в своем составе анализатор SAX. Он обладает высокой скоростью работы и потребляет совсем немного памяти при анализе XML. Но он основан на применении функций обратного вызова, поэтому для определенных частей данных, когда встречаются такие разделы документа XML, как открывающий и закрывающий теги, он просто вызывает определенные методы. Это означает, что вам придется задать обработчики для данных и самостоятельно отслеживать информацию о состоянии, что может оказаться далеко не простым делом. Это делает утверждение «simple» (простой) в названии «simple API for XML» не совсем соответствующим истине. Другой подход к обработке XML заключается

в использовании объектной модели документа (Document Object Model, DOM). В состав стандартной библиотеки языка Python входит и библиотека DOM XML. Как правило, анализатор DOM не отличается высокой скоростью работы и потребляет больше памяти, чем SAX, потому что он считывает дерево XML в память целиком и создает отдельные объекты для каждого узла дерева. Преимущество использования DOM заключается в том, что вам не придется отслеживать информацию о состоянии, так как каждый узел хранит информацию о родительских и дочерних узлах. Однако прикладной интерфейс DOM в лучшем случае приходится признать достаточно громоздким.

Имеется и третья возможность – ElementTree. ElementTree – это библиотека синтаксического анализа XML, которая входит в состав стандартной библиотеки языка Python, начиная с версии Python 2.5. Библиотеку ElementTree можно представить себе, как легковесный анализатор DOM, с простым и удобным прикладным интерфейсом. В дополнение к простоте и удобству в использовании этот анализатор потребляет незначительный объем памяти. Мы настоятельно рекомендуем использовать ElementTree. Если у вас возникнет потребность выполнять синтаксический анализ документов XML, попробуйте сначала воспользоваться библиотекой ElementTree.

Чтобы с помощью ElementTree приступить к анализу файла в формате XML, достаточно просто импортировать библиотеку и передать требуемый файл функции `parse()`:

```
In [1]: from xml.etree import ElementTree as ET

In [2]: tcusers = ET.parse('/etc/tomcat5.5/tomcat-users.xml')

In [3]: tcusers

Out[3]: <xml.etree.ElementTree.ElementTree instance at 0xabb4d0></xml>
```

Здесь, чтобы сократить объем ввода с клавиатуры при работе с библиотекой, мы импортировали модуль ElementTree под именем `ET`. Далее, мы предложили библиотеке выполнить разбор XML-файла со списком пользователей, полученного от механизма сервлетов Tomcat. Объект, созданный библиотекой ElementTree, мы назвали `tcusers`. Объект `tcusers` имеет тип `xml.etree.ElementTree.ElementTree`.

Мы удалили из файла пользователей сервера Tomcat примечания о порядке использования и текст лицензионного соглашения, в результате он принял следующий вид:

```
<?xml version="1.0" encoding="UTF-8"?>
<tomcat-users>
  <user name="tomcat" password="tomcat" roles="tomcat" />
  <user name="role1" password="tomcat" roles="role1" />
  <user name="both" password="tomcat" roles="tomcat,role1" />
</tomcat-users>
```

Во время разбора XML-файла метод parse() из библиотеки ElementTree создает и возвращает объект дерева, ссылка на который записывается в переменную tcusers. После этого данная переменная может использоваться для организации доступа к различным узлам дерева в файле XML. Наибольший интерес для нас представляют два метода этого объекта: find() и findall(). Метод find() отыскивает первый узел, соответствующий запросу, который ему передается, и возвращает объект Element, представляющий этот узел. Метод findall() отыскивает все узлы, соответствующие запросу, и возвращает список объектов Element, которые представляют эти узлы.

Перечень шаблонов, которые можно передавать методам find() и findall(), ограничен подмножеством выражений на языке XPath. В качестве критериев поиска можно указывать имя тега, символ «*», соответствующий всем дочерним элементам; символ «.», соответствующий текущему узлу; и комбинацию «//», соответствующую всем подчиненным узлам, начиная от точки поиска. Символ слеша (/) может использоваться в качестве разделителя критериев поиска. С помощью метода find() и имени тега мы попробовали отыскать первый узел user в файле пользователей Tomcat:

```
In [4]: first_user = tcusers.find('/user')

In [5]: first_user

Out[5]: <Element user at abdd88>
```

Мы передали методу find() критерий "/user". Начальный символ слеша указывает на абсолютный путь с началом в корневом узле. Текст 'user' определяет имя тега, который требуется отыскать. Отсюда следует, что метод find() вернет первый узел с тегом user. Здесь видно, что объект с именем first_user принадлежит к типу Element.

В число наиболее интересных для нас методов и атрибутов объекта Element входят attrib, find(), findall(), get(), tag и text. Атрибут attrib – это словарь атрибутов, принадлежащих данному объекту Element. Методы find() и findall() этого объекта работают точно так же, как одноименные методы объекта ElementTree. Метод get() используется для извлечения указанного атрибута из словаря атрибутов текущего тега XML. Атрибут tag содержит имя тега текущего объекта Element. Атрибут text содержит текст, расположенный в текстовом узле текущего объекта Element.

Ниже приводится элемент документа XML, соответствующий объекту first_user:

```
<user name="tomcat" password="tomcat" roles="tomcat" />
```

Теперь попробуем обратиться к методам и атрибутам объекта tcusers:

```
In [6]: first_user.attrib

Out[6]: {'name': 'tomcat', 'password': 'tomcat', 'roles': 'tomcat'}
```

```
In [7]: first_user.get('name')

Out[7]: 'tomcat'

In [8]: first_user.get('foo')

In [9]: first_user.tag

Out[9]: 'user'

In [10]: first_user.text
```

Теперь, когда вы получили некоторое представление о возможностях библиотеки ElementTree, рассмотрим более сложный пример. Мы выполним разбор файла пользователей Tomcat и отыщем все узлы `user`, где значение атрибута `name` соответствует значению, заданному нами (в данном случае `'tomcat'`), как показано в примере 3.27.

Пример 3.27. Разбор файла пользователей Tomcat с помощью библиотеки ElementTree

```
#!/usr/bin/env python

from xml.etree import ElementTree as ET

if __name__ == '__main__':
    infile = '/etc/tomcat5.5/tomcat-users.xml'
    tomcat_users = ET.parse(infile)
    for user in [e for e in tomcat_users.findall('/user') if
        e.get('name') == 'tomcat']:
        print user.attrib
```

Единственное, что представляет сложность в этом примере, – это использование генератора списков для поиска соответствующих атрибутов `name`. Этот сценарий возвращает следующий реультат:

```
jmjones@dinkgutsy:code$ python elementtree_tomcat_users.py
{'password': 'tomcat', 'name': 'tomcat', 'roles': 'tomcat'}
```

В заключение ниже приводится пример использования библиотеки ElementTree для извлечения некоторой информации из неудачно сформированного фрагмента XML. В операционной системе Mac OS X имеется утилита с именем `system_profiler`, которая отображает информацию о системе. Формат XML является одним из выходных форматов, которые поддерживает утилита `system_profiler`, но похоже, что поддержка формата XML была добавлена в самый последний момент. Мы предполагаем извлечь информацию о версии операционной системы, которая содержится в следующем фрагменте файла XML:

```
<dict>
    <key>_dataType</key>
    <string>SPSoftwareDataType</string>
    <key>_detailLevel</key>
    <integer>-2</integer>
    <key>_items</key>
```

```
    <array>
        <dict>
            <key>_name</key>
            <string>os_overview</string>
            <key>kernel_version</key>
            <string>Darwin 8.11.1</string>
            <key>os_version</key>
            <string>Mac OS X 10.4.11 (8S2167)</string>
        </dict>
    </array>
```

Вы спросите, почему на наш взгляд этот фрагмент XML оформлен неудачно? Дело в том, что ни в одном из тегов XML нет ни одного атрибута. В основной своей массе теги представляют типы данных. И такие теги с переменными значениями, как `key` и `string`, заключены в один и тот же родительский тег. Взгляните на пример 3.28.

Пример 3.28. Разбор файла, полученного в результате вызова утилиты system_profiler в Mac OS X

```
#!/usr/bin/env python
import sys

from xml.etree import ElementTree as ET
e = ET.parse('system_profiler.xml')

if __name__ == '__main__':
    for d in e.findall('/array/dict'):
        if d.find('string').text == 'SPSoftwareDataType':
            sp_data = d.find('array').find('dict')
            break
    else:
        print "SPSoftwareDataType NOT FOUND"
        sys.exit(1)

record = []
for child in sp_data.getchildren():
    record.append(child.text)
    if child.tag == 'string':
        print "%-15s -> %s" % tuple(record)
        record = []
```

Сценарий отыскивает все теги `dict`, в которых имеется дочерний элемент `string` с текстом `'SPSoftwareataType'`. Информация, которую требуется извлечь, находится в этом узле. В этом примере используется единственный метод, который не обсуждался ранее, – это метод `getchildren()`. Он просто возвращает список дочерних узлов указанного элемента. Кроме того, этот пример достаточно ясен, хотя сам файл XML можно было бы оформить лучше. Ниже приводится результат, полученный от сценария, когда он был запущен на ноутбуке, работающем под управлением операционной системы Mac OS X Tiger:

```
dink:~/code jmjones$ python elementtree_system_profile.py
_name           -> os_overview
kernel_version  -> Darwin 8.11.1
os_version      -> Mac OS X 10.4.11 (8S2167)
```

Библиотека стала прекрасным дополнением к стандартной библиотеке языка Python. Мы долгое время пользуемся ею и рады, что у нас есть такая возможность. Вы можете попробовать пользоваться библиотеками SAX и DOM, имеющимися в стандартной библиотеке языка Python, но мы думаем, что рано или поздно вы вернетесь к библиотеке ElementTree.

В заключение

В этой главе были обозначены некоторые фундаментальные принципы обработки текста в языке Python. Мы имели дело со встроенным типом `string`, с регулярными выражениями, стандартным вводом и выводом, с модулями `StringIO` и `urllib` из стандартной библиотеки. После этого использовали все полученные знания в двух примерах анализа файлов журналов веб-сервера Apache. В заключение были рассмотрены основы применения библиотеки ElementTree и продемонстрированы два примера использования для решения практических задач.

Складывается впечатление, что большинство специалистов по операционной системе UNIX, когда речь заходит об обработке текста более сложной, чем позволяют `grep` и `awk`, видят единственную альтернативу – язык Perl. Хотя Perl представляет собой очень мощный язык программирования, особенно в области обработки текста, мы полагаем, что язык Python может предложить ничуть не меньше возможностей. Фактически, особенно если учесть чистоту синтаксиса и простоту, с какой можно перейти от процедурного к объектно-ориентированному стилю программирования, мы считаем, что язык Python обладает преимуществами перед языком Perl. Поэтому мы надеемся, что в следующий раз, когда вам придется столкнуться с необходимостью реализовать обработку текста, вы сначала вспомните о языке Python.

4

Создание документации и отчетов

С нашей точки зрения, одним из самых утомительных и наименее желательных аспектов работы системного администратора является сбор различной информации и составление документации для пользователей. Эта работа может нести прямую выгоду вашим пользователям, которые будут читать документацию, или, возможно, косвенную выгоду пользователям, потому что вы или человек, пришедший вам на замену, сможете обратиться к ней при необходимости внести изменения в будущем. В любом случае создание документации является критически важным аспектом вашей деятельности. Но если это не та работа, выполнением которой вам хотелось бы заняться, тогда вы, скорее всего, отложите ее. Выполнить эту работу вам поможет Python. Нет, Python не напишет за вас документацию, но он поможет собрать, отформатировать и отправить документацию заинтересованным лицам.

В этой главе мы сосредоточим все свое внимание на следующих темах: сбор, форматирование и передача информации о написанных вами программах. Любая информация, которой вы предполагаете поделиться, где-то существует: она может находиться в файлах журналов, у вас в голове, она может быть доступна в виде результатов выполнения некоторых команд, она может даже находиться где-нибудь в базе данных. Самое первое, что необходимо сделать, это собрать необходимую информацию. Следующий шаг на пути к передаче информации другим людям заключается в том, чтобы оформить собранную информацию. Для оформления можно использовать такие форматы, как PDF, PNG, JPG, HTML или даже обычный текст. Наконец, необходимо передать эту информацию людям, которые заинтересованы в ней. Надо понять, каким образом заинтересованным лицам будет удобнее получать требуемую информацию: по электронной почте, на веб-сайте или просматривая файлы на совместно используемом диске.

Автоматизированный сбор информации

Первый шаг на пути к совместному использованию информации заключается в том, чтобы собрать ее. В этой книге имеются две главы, где рассматриваются способы сбора информации: «Текст» (глава 3) и «SNMP» (глава 7). В третьей главе содержатся примеры, демонстрирующие различные способы анализа и извлечения данных из текста. В частности, в одном из примеров этой главы из файлов журналов веб-сервера Apache извлекаются IP-адреса клиентов, количество переданных байтов каждому клиенту и код состояния протокола HTTP. В главе 7 имеются примеры выполнения запросов к системе на получение самой разнообразной информации, начиная от объема ОЗУ до пропускной способности сетевых интерфейсов.

Сбор информации может оказаться более сложным делом, чем простой поиск и извлечение определенных данных. Часто этот процесс может оказаться связанным с получением информации из одного представления, например, из файла журнала веб-сервера Apache, и сохранением ее в некотором промежуточном виде для последующего использования. Например, если представить, что вам необходимо создать диаграмму, показывающую количество байтов, загруженных каждым отдельным клиентом с уникальным IP-адресом за месяц с определенного веб-сервера Apache, тогда процесс сбора информации мог бы включать в себя ежедневный анализ файла журнала веб-сервера Apache, в ходе которого извлекается необходимая информация (в данном случае: IP-адрес и «количество переданных байтов» для каждого запроса), и сохранение ее в некотором хранилище данных, откуда потом ее можно будет получить. Примерами таких хранилищ данных могут служить реляционные базы данных, объектные базы данных, файлы-хранилища объектов, файлы в формате CSV и обычные текстовые файлы.

В оставшейся части этого раздела мы попробуем объединить некоторые понятия из глав, посвященных обработке текста и хранению данных. В частности, здесь будет показано, как соединить вместе приемы извлечения данных из главы 3 с приемами сохранения данных, обсуждаемыми в главе 12. При этом мы будем использовать те же библиотеки, что описывались в главе, касающейся вопросов обработки текста. Мы также будем использовать модуль `shelve`, который будет представлен в главе 12, для сохранения информации об HTTP-запросах, поступающих от каждого конкретного клиента.

Ниже приводится простой модуль, в котором используются модуль анализа файлов журналов веб-сервера Apache, созданный в предыдущей главе, и модуль `shelve`:

```
#!/usr/bin/env python

import shelve
import apache_log_parser_regex
```

```
logfile = open('access.log', 'r')
shelve_file = shelve.open('access.s')

for line in logfile:
    d_line = apache_log_parser_regex.dictify_logline(line)
    shelve_file[d_line['remote_host']] = \
        shelve_file.setdefault(d_line['remote_host'], 0) + \
        int(d_line['bytes_sent'])

logfile.close()
shelve_file.close()
```

В этом примере сначала импортируются модули `shelve` и `apache_log_parser_regex`. Модуль `shelve` входит в состав стандартной библиотеки языка Python. Модуль `apache_log_parser_regex` – это модуль, который был создан нами в главе 3. Затем открываются файл журнала веб-сервера Apache `access.log` и файл, куда будет сохраняться извлеченная информация, `access.s`. Далее выполняется обход всех строк в файле журнала и с помощью модуля разбора журнала для каждой строки создается словарь. Словарь содержит код состояния запроса HTTP, IP-адрес клиента и число байтов, переданных клиенту. После этого мы прибавляем число байтов для данного запроса к общему числу байтов, которое уже было сохранено ранее в объекте `shelve` для данного IP-адреса. Если в объекте `shelve` еще отсутствует запись для указанного IP-адреса, общее число переданных байтов автоматически устанавливается равным нулю. После обхода всех строк в файле журнала мы закрываем этот файл, а также объект `shelve`. Этот пример будет использоваться далее в этой главе, когда мы подойдем к вопросу форматирования информации.

Прием электронной почты

Вы наверное даже подумать не могли, что электронная почта может играть роль средства сбора информации и, тем не менее, это так. Представьте, что у вас имеется несколько серверов, ни один из которых не может соединяться с другими, но каждый из них обладает возможностью отправлять сообщения электронной почты. При наличии сценария, выполняющего мониторинг веб-приложений путем подключения к ним каждые несколько минут, можно было бы в этом случае использовать электронную почту как механизм передачи информации. В случае удачного или неудачного подключения можно было бы отправлять сообщения электронной почты с информацией об успехе или неудаче. И эти сообщения можно было бы использовать для составления отчетов – с целью предупредить ответственное лицо в случае появления проблем.

Для получения сообщений от сервера электронной почты обычно используются два протокола: IMAP и POP3. В стандартной поставке Python, куда «входят батарейки», имеются модули, поддерживающие оба эти протокола.

Из этих двух протоколов, пожалуй, наиболее часто используется протокол POP3 и доступ к электронной почте через этот протокол легко можно организовать с помощью модуля `poplib`. В примере 4.1 демонстрируется программный код, использующий модуль `poplib` для получения всех сообщений, хранящихся на указанном сервере, и записывающий их в отдельные файлы на диске.

Пример 4.1. Получение электронной почты по протоколу POP3

```
#!/usr/bin/env python

import poplib

username = 'someuser'
password = 'S3Cr37'

mail_server = 'mail.somedomain.com'

p = poplib.POP3(mail_server)
p.user(username)
p.pass_(password)
for msg_id in p.list()[1]:
    print msg_id
    outf = open('%s.eml' % msg_id, 'w')
    outf.write('\n'.join(p.retr(msg_id)[1]))
    outf.close()
p.quit()
```

Как видите, в этом примере для начала определяются `username` (имя пользователя), `password` (пароль) и `mail_server` (сервер электронной почты). Затем выполняется подключение к серверу, которому передаются предопределенные имя пользователя и пароль. Предположим, что соединение было выполнено успешно, и мы получили возможность просматривать электронную почту для данной учетной записи. После этого в цикле выполняется обход списка сообщений, извлечение и запись этих сообщений в файлы на диске. Единственное, что не предусмотрено в этом сценарии – сообщения не удаляются с сервера после их получения. Чтобы удалить эти сообщения, достаточно было бы добавить в сценарий вызов метода `dele()` после `retr()`.

Работа с протоколом IMAP реализуется почти так же просто, как и с протоколом POP3, но этот протокол не так хорошо описан в документации к стандартной библиотеке языка Python. В примере 4.2 приводится программный код, который выполняет те же самые действия, что и предыдущий пример, но с использованием протокола IMAP.

Пример 4.2. Получение электронной почты по протоколу IMAP

```
#!/usr/bin/env python

import imaplib

username = 'some_user'
password = '70P53Cr37'

mail_server = 'mail_server'
```

```
i = imaplib.IMAP4_SSL(mail_server)
print i.login(username, password)
print i.select('INBOX')
for msg_id in i.search(None, 'ALL')[1][0].split():
    print msg_id
    outf = open('%s.eml' % msg_id, 'w')
    outf.write(i.fetch(msg_id, '(RFC822)')[1][0][1])
    outf.close()
i.logout()
```

Как и в предыдущем примере, здесь также в самом начале сценария определяются `username` (имя пользователя), `password` (пароль) и `mail_server` (сервер электронной почты). Затем выполняется подключение к серверу IMAP через SSL. Затем выполняется вход и выбор папки электронной почты `INBOX`. Затем начинается обход всего, что будет найдено в папке. Метод `search()` плохо описан в документации к стандартной библиотеке языка Python. Этот метод имеет два обязательных аргумента – набор символов и критерий поиска. Какой набор символов является допустимым? В каком формате он должен указываться? Какие критерии поиска могут использоваться? Как правильно оформляется критерий? Мы можем, конечно, предполагать, что чтение IMAP RFC окажется полезным, но, к счастью, в примере использования протокола IMAP имеется достаточно информации, позволяющей организовать извлечение всех сообщений, хранящихся в папке. На каждой итерации цикла выполняется запись содержимого сообщения на диск. Следует заметить, что при этом все сообщения в папке будут помечены как «прочитанные». Для вас это может и не представлять большой проблемы, проблема была бы гораздо большей, если бы сообщения удалялись, но вам следует знать об этой особенности.

Сбор информации вручную

Рассмотрим также более сложный способ – сбор информации вручную. Здесь подразумевается информация, которая собирается вами путем просмотра и ввода вручную. В качестве примеров можно привести список серверов с соответствующими им IP-адресами и описанием функций, список контактов с адресами электронной почты, номерами телефонов и псевдонимами IM или список с датами отпусков членов вашей команды. Есть, конечно, инструменты, способные управлять, если не всей, то большей частью такой информации. Список серверов можно хранить в файлах Excel или OpenOffice Spreadsheet. Контакты можно хранить в Outlook или AddressBook.app. А расписание отпусков можно хранить как в Excel/OpenOffice Spreadsheet, так и в Outlook. Применение таких инструментов может стать решением в ситуациях, когда технологии свободно доступны, исходные данные могут представлять собой простой текст, а инструменты обеспечивают легко настраиваемый вывод информации и поддерживают формат HTML (или, что еще лучше, XHTML).

ПОРТРЕТ ЗНАМЕНИТОСТИ: ПАКЕТ RESTLESS

Аарон Хиллегасс (Aaron Hillegass)

Аарон Хиллегасс, работавший в компаниях NeXT и Apple, является экспертом в области разработки приложений для операционной системы Mac. Он является автором книги «Cocoa Programming for Mac OS X» (Big Nerd Ranch) и преподает программирование на платформе Cocoa в компании Big Nerd Ranch.

Загрузите полные исходные тексты пакета ReSTless из репозитария с примерами программного кода к этой книге по адресу: *http://www.oreilly.com/9780596515829*. Ниже показано, как вызвать сценарий на языке Python из вымышленного Cocoa-приложения:

```
#import "MyDocument.h"

@implementation MyDocument

- (id)init
{
    if (![super init]) {
        return nil;
    }

    // Что должно быть получено в случае нового документа
    textStorage = [[NSTextStorage alloc] init];
    return self;
}

- (NSString *)windowNibName
{
    return @"MyDocument";
}

- (void)prepareEditView
{
    // Менеджер размещения следит за хранилищем текста
    // и размещает текст в текстовом поле
    NSLayoutManager *lm = [editView layoutManager];

     // Отсоединить прежнее хранилище текста
     [[editView textStorage] removeLayoutManager:lm];

     // Присоединить новое хранилище текста
     [textStorage addLayoutManager:lm];
}

- (void)windowControllerDidLoadNib:(NSWindowController *) aController
{
```

```
    [super windowControllerDidLoadNib:aController];

    // Отобразить содержимое хранилища текста в текстовом поле
    [self prepareEditView];
}

#pragma mark Сохранение и загрузка

// Сохранение (URL всегда имеет тип file:)
- (BOOL)writeToURL:(NSURL *)absoluteURL
            ofType:(NSString *)typeName
            error:(NSError **)outError;
{
    return [[textStorage string] writeToURL:absoluteURL
                                 atomically:NO
                                   encoding:NSUTF8StringEncoding
                                      error:outError];
}

// Чтение (URL всегда имеет тип file:)
- (BOOL)readFromURL:(NSURL *)absoluteURL
             ofType:(NSString *)typeName
             error:(NSError **)outError
{
    NSString *string = [NSString stringWithContentsOfURL:absoluteURL
                                                encoding:NSUTF8StringEncoding
                                                    error:outError];
    // Ошибка чтения?
    if (!string) {
        return NO;
    }
    [textStorage release];
    textStorage = [[NSTextStorage alloc] initWithString:string
                                             attributes:nil];
    // Это возврат?
    if (editView) {
        [self prepareEditView];
    }

    return YES;
}

#pragma mark Создание и сохранение HTML

- (NSData *)dataForHTML
{
    // Создать задачу для запуска rst2html.py
    NSTask *task = [[NSTask alloc] init];

    // Предполагаемое местонахождение программы
    NSString *path = @"/usr/local/bin/rst2html.py";

    // Файл отсутствует? Попробовать отыскать внутри платформы python
```

```
    if (![[NSFileManager defaultManager] fileExistsAtPath:path]) {
        path = @"/Library/Frameworks/Python.framework/Versions/
                                    /Current/bin/rst2html.py;
    }
    [task setLaunchPath:path];

    // Подключить канал для ввода ReST
    NSPipe *inPipe = [[NSPipe alloc] init];
    [task setStandardInput:inPipe];
    [inPipe release];

    // Подключить канал для вывода HMTL
    NSPipe *outPipe = [[NSPipe alloc] init];
    [task setStandardOutput:outPipe];
    [outPipe release];

    // Запустить процесс
    [task launch];

    // Получить данные из текстового поля
    NSData *inData = [[textStorage string] dataUsingEncoding:
                                            NSUTF8StringEncoding];

    // Передать данные в канал и закрыть его
    [[inPipe fileHandleForWriting] writeData:inData];
    [[inPipe fileHandleForWriting] closeFile];

    // Прочитать данные из канала
    NSData *outData = [[outPipe fileHandleForReading]
                                            readDataToEndOfFile];

    // Завершить задачу
    [task release];

    return outData;
}

- (IBAction)renderRest:(id)sender
{
    // Запустить индикатор, чтобы пользователь видел,
    // что идет обработка
    [progressIndicator startAnimation:nil];

    // Получить html в виде NSData
    NSData *htmlData = [self dataForHTML];

    // Выдать html в основной WebFrame
    WebFrame *wf = [webView mainFrame];
    [wf loadData:htmlData
        MIMEType:@"text/html"
        textEncodingName:@"utf-8"
        baseURL:nil];

    // Остановить индикатор, чтобы пользователь видел,
    // что обработка закончена
```

```
    [progressIndicator stopAnimation:nil];
}

// Вызывается при выборе пункта меню
- (IBAction)startSavePanelForHTML:(id)sender
{
    // Куда сохранять по умолчанию?
    NSString *restPath = [self fileName];
    NSString *directory = [restPath stringByDeletingLastPathComponent];
    NSString *filename = [[[restPath lastPathComponent]
                           stringByDeletingPathExtension]
                           stringByAppendingPathExtension:@"html"];

    // Запустить диалог сохранения
    NSSavePanel *sp = [NSSavePanel savePanel];
    [sp setRequiredFileType:@"html"];
    [sp setCanSelectHiddenExtension:YES];
    [sp beginSheetForDirectory:directory
        file:filename
        modalForWindow:[editView window]
        modalDelegate:self
        didEndSelector:@selector(htmlSavePanel:endedWithCode:context:)
        contextInfo:NULL];
}

// Вызывается при закрытии диалога сохранения
- (void)htmlSavePanel:(NSSavePanel *)sp
                      endedWithCode:(int)returnCode
                      context:(void *)context
{
    // Пользователь щелкнул на кнопке Cancel?
    if (returnCode != NSOKButton) {
        return;
    }

    // Получить выбранное имя файла
    NSString *savePath = [sp filename];

    // Получить данные в формате HTML
    NSData *htmlData = [self dataForHTML];

    // Записать в файл
    NSError *writeError;
    BOOL success = [htmlData writeToFile:savePath
                    options:NSAtomicWrite
                    error:&writeError];

    // Записать не удалось?
    if (!success) {
        // Показать пользователю причину
        NSAlert *alert = [NSAlert alertWithError:writeError];
```

```
        [alert beginSheetModalForWindow:[editView window]
                        modalDelegate:nil
                        didEndSelector:NULL
                        contextInfo:NULL];
        return;
    }
}

#pragma mark Поддержка печати

- (NSPrintOperation *)printOperationWithSettings:(NSDictionary *)
                      printSettings error:(NSError **)outError
{
    // Получить информацию о параметрах настройки печати
    NSPrintInfo *printInfo = [self printInfo];

    // Получить поле, где отображается весь документ HTML
    NSView *docView = [[[webView mainFrame] frameView] documentView];

    // Создать задание печати
    return [NSPrintOperation printOperationWithView:docView
            printInfo:printInfo];
}

@end
```

Несмотря на наличие различных альтернатив, мы собираемся предложить здесь текстовый формат, который называется reStructuredText (или reST). Вот как описывается формат reStructuredText на веб-сайте:

> reStructuredText – это легкая для чтения, обеспечивающая отображение текста в режиме «что видишь, то и получаешь» простая текстовая разметка и одновременно система синтаксического анализа. Ее удобно использовать для встроенной документации к программам (например, в строках документирования языка Python), для быстрого создания веб-страниц, для создания самостоятельных документов. Разметка reStructuredText предусматривает возможность расширения под нужды различных прикладных областей. Синтаксический анализатор reStructuredText является составной частью пакета Docutils. Разметка reStructuredText представляет собой пересмотренную реализацию легковесных систем разметки StructuredText и Setext.

Формат ReST считается предпочтительным форматом для создания документации в языке Python. Если вы создали программный пакет на языке Python и решили выложить его в репозитарий PyPI, то от вас будут ожидать, что сопроводительная документация к пакету будет иметь формат ReST. Многие самостоятельные проекты, использую-

щие язык Python, применяют формат ReST в качестве основного для оформления своей документации.

Итак, какими же преимуществами обладает ReST как формат, используемый для создания документации? Во-первых, он достаточно прост. Во-вторых, знакомство с разметкой происходит практически сразу. Как только перед вами оказывается структура документа, вы быстро начинаете понимать, что имел в виду автор. Ниже приводится очень простой пример файла в формате ReST:

```
=======
Heading
=======
SubHeading
----------
This is just a simple
little subsection. Now,
we'll show a bulleted list:

- item one
- item two
- item three
```

Этот пример позволяет без чтения документации представить, как выглядит правильно оформленный файл в формате reStructuredText. Возможно, при этом вы еще не в состоянии создать текстовый файл в формате ReST, но, по крайней мере, вы сможете читать его.

В-третьих, преобразование документов из формата ReST в формат HTML выполняется очень просто. И на этом третьем пункте мы сосредоточим свое внимание в этом разделе. Мы не будем пытаться представить здесь учебник по reStructuredText. Если вам захочется ознакомиться с синтаксисом разметки, посетите страницу *http://docutils.sourceforge.net/docs/user/rst/quickref.html*.

Мы пройдем все этапы преобразования разметки ReST в HTML, используя документ, который мы только что показали в качестве примера:

```
In [2]: import docutils.core

In [3]: rest = '''=======
   ...: Heading
   ...: =======
   ...: SubHeading
   ...: ----------
   ...: This is just a simple
   ...: little subsection. Now,
   ...: we'll show a bulleted list:
   ...:
   ...: - item one
   ...: - item two
   ...: - item three
   ...: '''
```

```
In [4]: html = docutils.core.publish_string(source=rest, writer_name='html')

In [5]: print html[html.find('<body>') + 6:html.find('</body>')]

<div class="document" id="heading">
<h1 class="title">Heading</h1>
<h2 class="subtitle" id="subheading">SubHeading</h2>
<p>This is just a simple
little subsection. Now,
we'll show a bulleted list:</p>
<ul class="simple">
<li>item one</li>
<li>item two</li>
<li>item three</li>
</ul>
</div>
```

Это оказалось совсем несложно. Мы импортировали модуль `docutils.core`. Затем определили строку, содержащую текст в формате reStructuredText, передали эту строку методу `docutils.core.publish_string()` и потребовали от него преобразовать строку в формат HTML. Затем с помощью операции извлечения среза мы извлекли текст, заключенный между тегами `<body>` и `</body>`. Мы извлекли срез потому, что библиотека docutils, использованная здесь для преобразования текста в формат HTML, вставляет в страницу HTML, созданную с ее помощью, каскадные таблицы стилей, чтобы она не выглядела слишком уныло.

Теперь, когда вы увидели, насколько все просто, рассмотрим другой пример, который находится ближе к системному администрированию. Любой хороший сисадмин должен помнить перечень своих серверов и задачи, которые они решают. Поэтому ниже приводится пример, показывающий, как можно составить список серверов сначала в простом текстовом виде, а затем преобразовать его в формат HTML:

```
In [6]: server_list = '''============== ============ ================
   ...: Server Name IP Address Function
   ...: ============== ============ ================
   ...: card 192.168.1.2 mail server
   ...: vinge 192.168.1.4 web server
   ...: asimov 192.168.1.8 database server
   ...: stephenson 192.168.1.16 file server
   ...: gibson 192.168.1.32 print server
   ...: ============== ============ ================'''

In [7]: print server_list
============== ============ ================
  Server Name   IP Address   Function
============== ============ ================
card           192.168.1.2  mail server
vinge          192.168.1.4  web server
asimov         192.168.1.8  database server
stephenson     192.168.1.16 file server
```

```
gibson          192.168.1.32 print server
============== ============ ================

In [8]: html = docutils.core.publish_string(source=server_list,
writer_name='html')

In [9]: print html[html.find('<body>') + 6:html.find('</body>')]

<div class="document">
<table border="1" class="docutils">
<colgroup>
<col width="33%" />
<col width="29%" />
<col width="38%" />
</colgroup>
<thead valign="bottom">
<tr><th class="head">Server Name</th>
<th class="head">IP Address</th>
<th class="head">Function</th>
</tr>
</thead>
<tbody valign="top">
<tr><td>card</td>
<td>192.168.1.2</td>
<td>mail server</td>
</tr>
<tr><td>vinge</td>
<td>192.168.1.4</td>
<td>web server</td>
</tr>
<tr><td>asimov</td>
<td>192.168.1.8</td>
<td>database server</td>
</tr>
<tr><td>stephenson</td>
<td>192.168.1.16</td>
<td>file server</td>
</tr>
<tr><td>gibson</td>
<td>192.168.1.32</td>
<td>print server</td>
</tr>
</tbody>
</table>
</div>
```

Еще одна замечательная и простая текстовая разметка называется Textile. Согласно описанию на веб-сайте: «Textile получает текст с *простой* разметкой и воспроизводит корректный код разметки XHTML. Этот формат используется в веб-приложениях, в системах управления содержимым, программным обеспечением блогов и форумов». Если Textile является языком разметки, то почему он описывается в книге,

посвященной языку Python? Причина в том, что для языка Python существует библиотека, которая позволяет обрабатывать разметку Textile и преобразовывать ее в XHTML. Вы можете написать отдельную утилиту командной строки, которая с помощью библиотеки будет преобразовывать файлы из формата Textile в формат XHTML. Или можно вызывать модуль, выполняющий преобразование формата Textile, в некотором сценарии и программным способом обработать полученную разметку XHTML. В любом случае разметка Textile и модуль обработки Textile могут оказаться неплохим подспорьем в создании документации.

Установить модуль Textile можно с помощью команды `easy_install textile` или с помощью системы управления пакетами, имеющейся в вашем дистрибутиве. В Ubuntu, например, пакет называется `python-textile` и установить его можно командой `apt-get install python-textile`. Как только модуль Textile будет установлен, вы сможете приступить к его использованию, просто импортируя его, создавая объект `Textiler` и вызывая единственный метод этого объекта. Ниже приводится пример преобразования маркированного списка из формата Textile в формат XHTML:

```
In [1]: import textile

In [2]: t = textile.Textiler('''* item one
   ...: * item two
   ...: * item three''')

In [3]: print t.process()
<ul>
<li>item one</li>
<li>item two</li>
<li>item three</li>
</ul>
```

Мы не будем пытаться дать здесь описание формата Textile. В Сети имеется множество ресурсов с этой информацией. Например, по адресу *http://hobix.com/textile/* вы найдете отличное руководство по использованию Textile. Хотя мы и не собираемся слишком углубляться в описание формата Textile, тем не менее, мы посмотрим, как применить формат Textile для оформления информации, собранной вручную, – списка серверов с соответствующими им IP-адресами и функциями:

```
In [1]: import textile

In [2]: server_list = '''|_. Server Name|_. IP Address|_. Function|
   ...: |card|192.168.1.2|mail server|
   ...: |vinge|192.168.1.4|web server|
   ...: |asimov|192.168.1.8|database server|
   ...: |stephenson|192.168.1.16|file server|
   ...: |gibson|192.168.1.32|print server|'''

In [3]: print server_list
|_. Server Name|_. IP Address|_. Function|
```

```
|card|192.168.1.2|mail server|
|vinge|192.168.1.4|web server|
|asimov|192.168.1.8|database server|
|stephenson|192.168.1.16|file server|
|gibson|192.168.1.32|print server|

In [4]: t = textile.Textiler(server_list)

In [5]: print t.process()
<table>
<tr>
<th>Server Name</th>
<th>IP Address</th>
<th>Function</th>
</tr>
<tr>
<td>card</td>
<td>192.168.1.2</td>
<td>mail server</td>
</tr>
<tr>
<td>vinge</td>
<td>192.168.1.4</td>
<td>web server</td>
</tr>
<tr>
<td>asimov</td>
<td>192.168.1.8</td>
<td>database server</td>
</tr>
<tr>
<td>stephenson</td>
<td>192.168.1.16</td>
<td>file server</td>
</tr>
<tr>
<td>gibson</td>
<td>192.168.1.32</td>
<td>print server</td>
</tr>
</table>
```

Как видите, оба формата, ReST и Textile, могут эффективно использоваться в сценариях на языке Python для преобразования текстовых данных. Если у вас действительно имеются такие данные, как списки серверов и контактов, которые требуется преобразовывать в формат HTML и затем предпринимать какие-либо дополнительные действия (например, отправлять HTML по электронной почте или передавать HTML-страницы куда-нибудь на веб-сервер по протоколу FTP), то библиотека docutils или Textile может оказаться для вас полезным инструментом.

Форматирование информации

Следующий этап на пути передачи информации в руки тех, кому она необходима, заключается в форматировании данных так, чтобы их легко можно было воспринимать и понимать. Мы считаем, что информация должна быть представлена в том виде, в каком, по крайней мере, ее легко будет воспринимать, но еще лучше, если оформление будет еще и привлекательным. С технической точки зрения применение форматов ReST и Textile охватывает сразу оба этапа – сбора и форматирования информации, но в следующих примерах мы сосредоточимся исключительно на преобразовании уже собранных данных в более представительный вид.

Графические изображения

В следующих двух примерах мы продолжим пример анализа файла журнала веб-сервера Apache, из которого извлекаются IP-адреса клиентов и количество переданных байтов. В предыдущем разделе, продолжая этот пример, мы создали промежуточный файл, содержащий информацию, которой мы предполагаем поделиться с другими. Поэтому на основе этого промежуточного файла мы создадим диаграмму, чтобы эти данные было проще воспринимать:

```
#!/usr/bin/env python

import gdchart
import shelve

shelve_file = shelve.open('access.s')
items_list = [(i[1], i[0]) for i in shelve_file.items()]
items_list.sort()
bytes_sent = [i[0] for i in items_list]
#ip_addresses = [i[1] for i in items_list]
ip_addresses = ['XXX.XXX.XXX.XXX' for i in items_list]

chart = gdchart.Bar()
chart.width = 400
chart.height = 400
chart.bg_color = 'white'
chart.plot_color = 'black'
chart.xtitle = "IP Address"
chart.ytitle = "Bytes Sent"
chart.title = "Usage By IP Address"
chart.setData(bytes_sent)
chart.setLabels(ip_addresses)
chart.draw("bytes_ip_bar.png")

shelve_file.close()
```

В этом примере были импортированы два модуля, `gdchart` и `shelve`. Затем был открыт файл хранилища объектов, созданный в предыдущем примере. Объект `shelve` обладает тем же интерфейсом, что и встроен-

ный тип dictionary, поэтому имеется возможность вызвать его метод items(). Этот метод возвращает список кортежей, где первый элемент кортежа соответствует ключу словаря, а второй элемент – значению этого ключа. Применение метода items() обеспечивает возможность сортировки данных, что определенно будет иметь смысл, когда мы начнем рисовать диаграмму. Кроме того, с помощью генератора списков мы меняем порядок следования элементов в кортежах. То есть вместо кортежей с элементами (ip_address, bytes_sent) мы получаем кортежи (bytes_sent, ip_address). Затем выполняется сортировка кортежей в списке, а поскольку в каждом кортеже первым элементом является значение bytes_sent, метод list.sort() выполнит сортировку по этому элементу. Далее с помощью генератора списков извлекаются значения bytes_sent и ip_address. Обратите внимание, что мы прибегли к сокрытию IP-адресов, заменив их значением XXX.XXX.XXX.XXX, потому что данные для этого примера были получены нами из файла журнала действующего веб-сервера.

После выборки данных, на основе которых будет построена диаграмма, можно приступать к созданию ее графического представления, используя модуль gdchart. Сначала создается объект gdchart.Bar. Это простой объект диаграммы, в котором необходимо установить значения некоторых атрибутов, а затем с его помощью можно будет отобразить диаграмму в файл PNG. После этого определяются размеры диаграммы в пикселях, цвет фона и цвет переднего плана и создаются заголовки. Устанавливаются данные и метки для диаграммы, полученные в результате анализа файла журнала веб-сервера Apache. В заключение вызывается метод draw(), который выводит диаграмму в файл, и производится закрытие файла хранилища. Изображение полученной диаграммы показано на рис. 4.1.

Ниже приводится другой пример сценария, выполняющего визуализацию данных, находящихся в файле хранилища, но на этот раз программа создает не гистограмму, а круговую диаграмму:

```
#!/usr/bin/env python

import gdchart
import shelve
import itertools

shelve_file = shelve.open('access.s')
items_list = [(i[1], i[0]) for i in shelve_file.items() if i[1] > 0]
items_list.sort()
bytes_sent = [i[0] for i in items_list]
#ip_addresses = [i[1] for i in items_list]
ip_addresses = ['XXX.XXX.XXX.XXX' for i in items_list]

chart = gdchart.Pie()
chart.width = 800
chart.height = 800
chart.bg_color = 'white'
```

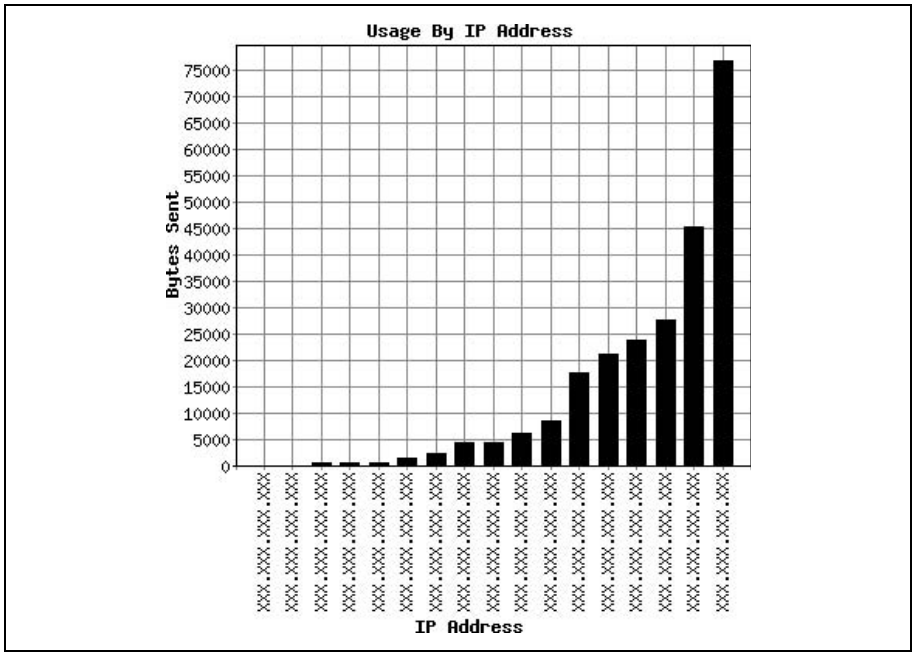


Рис. 4.1. *Гистограмма количества байтов, переданных по запросам с каждого IP-адреса*

```
color_cycle = itertools.cycle([0xDDDDDD, 0x111111, 0x777777])
color_list = []
for i in bytes_sent:
    color_list.append(color_cycle.next())
chart.color = color_list

chart.plot_color = 'black'
chart.title = "Usage By IP Address"
chart.setData(*bytes_sent)
chart.setLabels(ip_addresses)
chart.draw("bytes_ip_pie.png")

shelve_file.close()
```

Принцип действия этого сценария практически идентичен предыдущему за несколькими исключениями. Во-первых, в этом сценарии создается экземпляр объекта `gdchart.Pie`, а не `gdchart.Bar`. Во-вторых, мы определили отдельные цвета для каждого сектора диаграммы. Это круговая диаграмма и поэтому, если все сектора вывести черным цветом, такую диаграмму будет невозможно читать, в связи с чем нами было принято решение организовать чередование трех градаций серого цвета. Чередование этих трех цветов было реализовано с помощью

функции `cycle()` из модуля `itertools`. Мы рекомендуем вам обратить свое внимание на модуль `itertools`. В нем имеется значительное число интересных функций, которые помогут вам в работе с итерируемыми объектами (такими, как списки). Результат работы этого сценария, создающего круговую диаграмму, приводится на рис. 4.2.

Единственный недостаток круговой диаграммы состоит в том, что произошло наложение (сокрытие) IP-адресов, соответствующих секторам с минимальными значениями переданных байтов. Гистограммы и круговые диаграммы существенно облегчают восприятие данных, находящихся в файле хранилища, процесс создания диаграмм оказался на удивление простым, и включение данных было удивительно легким делом.

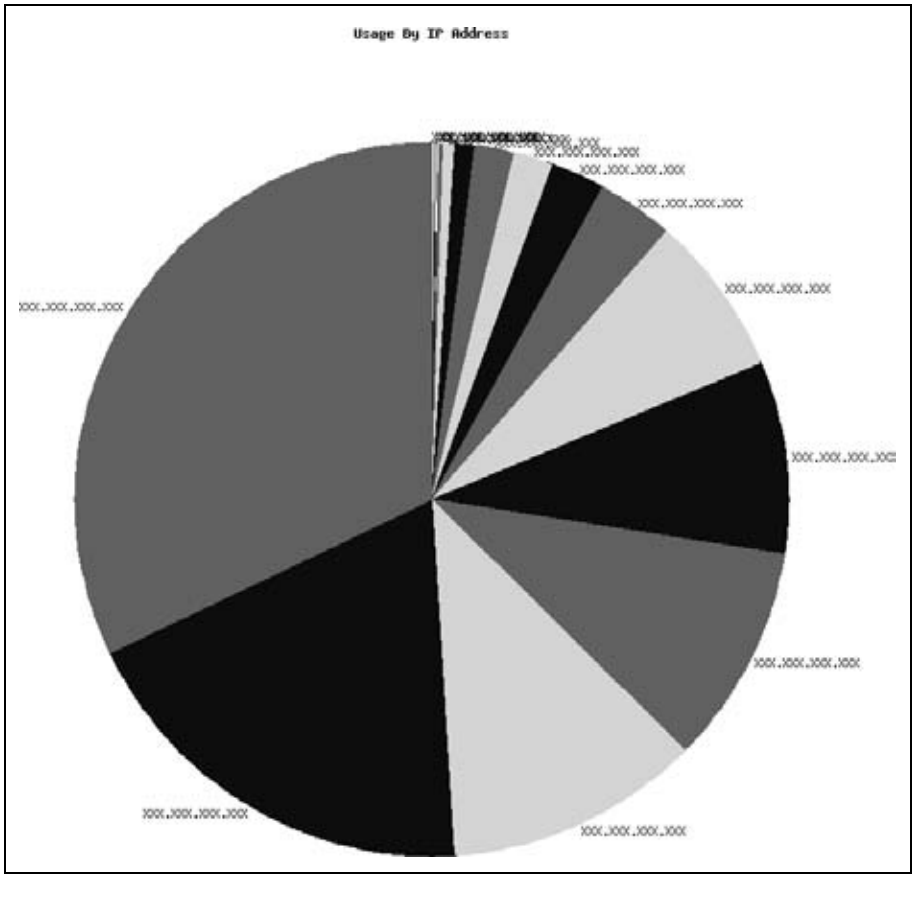


Рис. 4.2. *Круговая диаграмма с количеством байтов, переданных по запросам с каждого IP-адреса*

PDF

Другой способ представления информации из файлов с данными заключается в сохранении их в формате PDF. Формат PDF приобрел господствующее положение, и мы готовы предполагать, что все документы можно преобразовать в PDF. Знание и умение создавать документы в формате PDF может существенно облегчить жизнь вам как системным администраторам. После прочтения этого раздела вы сможете применять полученные знания для создания отчетов в формате PDF о загруженности сети, об учетных записях пользователей и так далее. Мы также опишем способ автоматического встраивания документов PDF в сообщения электронной почты с использованием языка Python.

Библиотеку ReportLab в мире библиотек, предназначенных для работы с форматом PDF, можно сравнить с 350-килограммовой гориллой. В документе, расположенном по адресу *http://www.reportlab.com/docs/userguide.pdf*, вы найдете значительное число примеров использования библиотеки ReportLab. Кроме этого раздела мы настоятельно рекомендуем вам ознакомиться с официальной документацией проекта ReportLab. Для установки библиотеки ReportLab в Ubuntu достаточно дать команду `apt-get install python-reportlab`. Если вы пользуетесь другим дистрибутивом, воспользуйтесь помощью менеджера пакетов в своей операционной системе. Иначе у вас всегда есть возможность получить дистрибутив с исходными текстами.

В примере 4.3 приводится пример использования библиотеки ReportLab для создания простого документа PDF «Hello World».

Пример 4.3. Документ PDF – «Hello World»

```
#!/usr/bin/env python

from reportlab.pdfgen import canvas
def hello():
    c = canvas.Canvas("helloworld.pdf")
    c.drawString(100,100,"Hello World")
    c.showPage()
    c.save()
hello()
```

Нам хотелось бы сделать несколько замечаний к процессу создания PDF-документа «Hello World». Во-первых, сначала был создан объект `canvas`. Далее был использован метод `drawString()`, который можно считать эквивалентом метода `file_obj.write()` в случае текстовых файлов. В заключение были вызваны метод `showPage()`, завершающий процесс рисования, и метод `save()`, который фактически создает файл PDF. Если выполнить этот сценарий, в результате будет получен пустой одностраничный документ PDF с надписью «Hello World» в самом низу.

Если вы загрузили дистрибутив с исходными текстами библиотеки ReportLab, вы можете воспользоваться тестами, которые были включены

как примеры оформления документации. То есть при запуске эти тесты создают комплект файлов PDF, которые можно изучить и посмотреть, как с помощью библиотеки ReportLab можно добиться различных визуальных эффектов.

Теперь, когда вы увидели, как с помощью библиотеки ReportLab создаются документы PDF, посмотрим, как с ее же помощью создать отчет об использовании дискового пространства. Такой отчет может оказаться весьма полезным. Взгляните на пример 4.4.

Пример 4.4. Отчет об использовании дискового пространства

```
#!/usr/bin/env python
import subprocess
import datetime
from reportlab.pdfgen import canvas
from reportlab.lib.units import inch

def disk_report():
    p = subprocess.Popen("df -h", shell=True,
                         stdout=subprocess.PIPE)
    return p.stdout.readlines()

def create_pdf(input,output="disk_report.pdf"):
    now = datetime.datetime.today()
    date = now.strftime("%h %d %Y %H:%M:%S")
    c = canvas.Canvas(output)
    textobject = c.beginText()
    textobject.setTextOrigin(inch, 11*inch)
    textobject.textLines('''
    Disk Capacity Report: %s
    ''' % date)
    for line in input:
        textobject.textLine(line.strip())
    c.drawText(textobject)
    c.showPage()
    c.save()

report = disk_report()
create_pdf(report)
```

Этот сценарий генерирует отчет с заголовком «Disk Capacity Report» (Отчет об использовании дискового пространства), отображающий текущую информацию об использовании дискового пространства, а также указывается дата и время создания отчета. Для сценария такого незначительного размера это очень неплохо. Рассмотрим некоторые особенности этого примера. Во-первых, функция `disk_report()` просто принимает вывод команды `df -h` и возвращает его в виде списка строк. Далее, в функции `create_pdf()` сначала создается надпись с текущей датой и временем. Самая важная часть этого примера – объект `textobject`.

Объект `textobject` создается, чтобы потом поместить его в документ PDF. Создание объекта `textobject` производится с помощью метода

`beginText()`. Затем определяется способ размещения данных на странице. Наш документ будет содержать страницы размером 8.5×11 дюймов, поэтому, чтобы поместить текст в самом верху страницы, мы сообщаем текстовому объекту, что текст будет находиться в 11 дюймах от начала координат. После этого создается заголовок выводом строки в текстовый объект, и мы завершаем работу, выполняя обход списка строк, полученных в результате работы команды `df`. Обратите внимание: здесь использован метод `line.strip()` для удаления символов новой строки. Если этого не сделать, символы новой строки будут присутствовать в документе в виде черных квадратов.

Имеется возможность создавать намного более сложные документы PDF, добавляя цвета и изображения, но обо всем этом вы сможете узнать во время чтения превосходного руководства пользователя, поставляемого вместе с библиотекой ReportLib. Главное, что следует из этих примеров, текст является основным объектом, хранящим данные, которые требуется отобразить.

Распространение информации

После того как данные будут собраны и отформатированы, необходимо передать их тем, кто заинтересован в их получении. В этом разделе мы сосредоточим основное внимание на передаче документации с помощью электронной почты. Если вам потребуется передать некоторую документацию на веб-сервер, где ее смогут увидеть ваши пользователи, вы сможете использовать для этого протокол FTP. Использование стандартного модуля Python для работы с протоколом FTP мы рассмотрим в следующей главе.

Передача электронной почты

Работа с электронной почтой является важной составляющей в деятельности системного администратора. Мало того, что нам приходится управлять почтовыми серверами, но нам часто приходится придумывать способы отправки предупреждений по электронной почте. Стандартная библиотека языка Python обладает потрясающей поддержкой возможности отправлять сообщения электронной почты, но в книгах об этом упоминается очень редко. Любой системный администратор должен иметь тщательно налаженный механизм автоматизированной отправки электронной почты, поэтому с этом разделе будет показано, как можно решать разнообразные задачи, связанные с электронной почтой, используя язык Python.

Передача простых сообщений

В состав Python входят два независимых друг от друга пакета, позволяющих отправлять сообщения по электронной почте. Один низкоуровневый пакет, smtplib, представляет собой интерфейс к протоколу

SMTP, отвечающий требованиям различных спецификаций RFC. Другой пакет, email, помогает выполнять анализ и создание сообщений электронной почты. В примере 4.5 с помощью средств пакета `smtplib` создается строка, представляющая тело сообщения, а затем с помощью пакета email производится его отправка серверу электронной почты.

Пример 4.5. Отправка сообщений по протоколу SMTP

```
#!/usr/bin/env python

import smtplib
mail_server = 'localhost'
mail_server_port = 25
from_addr = 'sender@example.com'
to_addr = 'receiver@example.com'

from_header = 'From: %s\r\n' % from_addr
to_header = 'To: %s\r\n\r\n' % to_addr
subject_header = 'Subject: nothing interesting'

body = 'This is a not-very-interesting email.'

email_message = '%s\n%s\n%s\n\n%s' % (from_header, to_header,
                                      subject_header, body)

s = smtplib.SMTP(mail_server, mail_server_port)
s.sendmail(from_addr, to_addr, email_message)
s.quit()
```

Здесь мы определили имя хоста и номер порта сервера электронной почты, а также адреса «to» (получатель) и «from» (отправитель). Затем производится сборка самого сообщения путем объединения заголовков с телом сообщения. В заключение выполняется подключение к серверу SMTP и производится отправка сообщения по адресу `to_addr` с адреса `from_addr`. Следует также заметить, что добавление комбинаций символов `\r\n` в поля `From:` и `To:` выполнено в соответствии с требованиями RFC.

В главе 10 в разделе «Планирование процессов Python» приводится пример программного кода на языке Python, который запускается как задание планировщика cron выполняющее отправку сообщений электронной почты. А теперь перейдем от этого простого примера к более сложным операциям с электронной почтой, которые можно реализовать на языке Python.

Аутентификация по протоколу SMTP

Наш последний пример был чрезвычайно прост, поскольку нет ничего сложного в том, чтобы реализовать отправку почты на языке Python, но, к сожалению, подавляющее большинство серверов SMTP вынудят вас проходить процедуру аутентификации, поэтому предыдущий пример в таких ситуациях окажется бесполезным. Порядок выполнения аутентификации демонстрируется в примере 4.6.

Пример 4.6. Аутентификация по протоколу SMTP

```
#!/usr/bin/env python
import smtplib
mail_server = 'smtp.example.com'
mail_server_port = 465

from_addr = 'foo@example.com'
to_addr = 'bar@exmaple.com'

from_header = 'From: %s\r\n' % from_addr
to_header = 'To: %s\r\n\r\n' % to_addr
subject_header = 'Subject: Testing SMTP Authentication'

body = 'This mail tests SMTP Authentication'

email_message = '%s\n%s\n%s\n\n%s' % (from_header, to_header,
                                      subject_header, body)

s = smtplib.SMTP(mail_server, mail_server_port)
s.set_debuglevel(1)
s.starttls()
s.login("fatalbert", "mysecretpassword")
s.sendmail(from_addr, to_addr, email_message)
s.quit()
```

Основное отличие от предыдущего примера заключается в том, что здесь указываются имя пользователя и пароль. Перед отправкой вызовом метода `debuglevel()` мы активировали режим отладки и затем запустили соединение SSL с использованием метода `starttls()`. Включение режима отладки перед прохождением аутентификации – это замечательная идея. Если взглянуть на отладочную информацию, полученную в случае неудачи, она будет иметь вид, как показано ниже:

```
$ python2.5 mail.py
send: 'ehlo example.com\r\n'
reply: '250-example.com Hello example.com [127.0.0.1], pleased to meet
you\r\n'
reply: '250-ENHANCEDSTATUSCODES\r\n'
reply: '250-PIPELINING\r\n'
reply: '250-8BITMIME\r\n'
reply: '250-SIZE\r\n'
reply: '250-DSN\r\n'
reply: '250-ETRN\r\n'
reply: '250-DELIVERBY\r\n'
reply: '250 HELP\r\n'
reply: retcode (250); Msg: example.com example.com [127.0.0.1], pleased to
meet you
ENHANCEDSTATUSCODES
PIPELINING
8BITMIME
SIZE
DSN
ETRN
```

```
DELIVERBY
HELP
send: 'STARTTLS\r\n'
reply: '454 4.3.3 TLS not available after start\r\n'
reply: retcode (454); Msg: 4.3.3 TLS not available after start
```

В этом примере сервер, с которым мы попытались установить соединение SSL, не поддерживает такую возможность. Можно без особого труда обойти эту и другие потенциальные проблемы, создавая сценарии, которые включают в себя обработку ошибок отправки электронной почты, реализуя попытки отправки через каскад серверов, вплоть до попытки отправить почту через локальный сервер.

Реализация отправки вложений на языке Python

Отправка сообщений, состоящих исключительно из текста, выполняется очень просто. Однако на языке Python можно реализовать отправку сообщений с использованием стандарта MIME, что означает возможность добавлять вложения в исходящие сообщения. В предыдущем разделе этой главы мы рассматривали возможность создания отчетов в формате PDF. Системные администраторы – нетерпеливые люди, поэтому мы опустим подробности о происхождении MIME и сразу же перейдем к отправке электронной почты с вложениями, как показано в примере 4.7.

Пример 4.7. Отправка документа PDF, вложенного в сообщение электронной почты

```
import email
from email.MIMEText import MIMEText
from email.MIMEMultipart import MIMEMultipart
from email.MIMEBase import MIMEBase
from email import encoders
import smtplib
import mimetypes

from_addr = 'noah.gift@gmail.com'
to_addr = 'jjinux@gmail.com'
subject_header = 'Subject: Sending PDF Attachemt'
attachment = 'disk_usage.pdf'
body = '''
This message sends a PDF attachment created with Report
Lab.
'''

m = MIMEMultipart()
m["To"] = to_addr
m["From"] = from_addr
m["Subject"] = subject_header

ctype, encoding = mimetypes.guess_type(attachment)
print ctype, encoding
maintype, subtype = ctype.split('/', 1)
```

```
print maintype, subtype

m.attach(MIMEText(body))
fp = open(attachment, 'rb')
msg = MIMEBase(maintype, subtype)
msg.set_payload(fp.read())
fp.close()
encoders.encode_base64(msg)
msg.add_header("Content-Disposition", "attachment", filename=attachment)
m.attach(msg)

s = smtplib.SMTP("localhost")
s.set_debuglevel(1)
s.sendmail(from_addr, to_addr, m.as_string())
s.quit()
```

Итак, мы совсем немного поколдовали, чтобы закодировать и отправить по электронной почте наш созданный ранее отчет в формате PDF об использовании дискового пространства.

Trac

Trac – это вики (wiki) и система отслеживания проблем. Она обычно используется в процессе разработки программного обеспечения и написана на языке Python, но в действительности может использоваться везде, где необходима вики или система регистрации сообщений. Последнюю версию документации к системе Trac можно найти по адресу: *http://trac.edgewall.org/*. Детальное обсуждение Trac выходит далеко за рамки этой книги, но это достаточно хороший инструмент, который может использоваться для регистрации поступающих сообщений об ошибках. Одна из интересных особенностей Trac состоит в том, что эта система допускает возможность расширения с помощью дополнительных модулей.

Мы упомянули эту систему, потому что она вписывается во все три последние темы, которые мы обсуждали: сбор информации, форматирование и распространение. Реализация вики в системе дает возможность пользователям создавать веб-страницы с помощью броузеров. Информация, которую они добавляют таким способом, становится доступна в формате HTML другим пользователям. Таким образом, система реализует полный цикл, обсуждаемый в этой главе.

Точно так же система регистрации и отслеживания сообщений дает возможность пользователям помещать свои предложения или сообщения об обнаруженных проблемах. Вы с ее помощью сможете составлять отчеты о сообщениях, введенных через веб-интерфейс, и даже генерировать отчеты в формате CSV. Напомним еще раз, что система Trac охватывает полный цикл от сбора до распространения информации, который рассматривается в этой главе.

Мы рекомендуем вам познакомиться с системой Trac поближе, чтобы понять, насколько полно она отвечает вашим потребностям. Может быть, вам потребуется нечто более мощное или наоборот, что-нибудь попроще, но эта система достойна того, чтобы познакомиться с ней поближе.

В заключение

В этой главе мы рассмотрели автоматизированный и ручной способы сбора информации. Мы также рассмотрели способы объединения собранных данных в документы наиболее распространенных форматов, а именно: HTML, PDF и PNG. В заключение мы рассмотрели способы передачи информации заинтересованным в ней лицам. Как мы уже говорили в начале главы, составление документации может быть не самой приятной частью вашей работы. Возможно, при поступлении на работу вы даже не представляли себе, что придется заниматься документацией. Но ясная и понятная документация – это чрезвычайно важный элемент системного администрирования. Мы надеемся, что советы из этой главы помогут сделать несколько рутинную работу по созданию документации намного более увлекательной.

5

Сети

Под сетями обычно подразумевается объединение множества компьютеров с целью обеспечить разного рода взаимодействия между ними. Однако нас в первую очередь интересуют не взаимодействия между компьютерами, а взаимодействия между процессами. При этом, с точки зрения приемов, которые мы собираемся продемонстрировать, совершенно неважно, выполняются взаимодействующие процессы на разных компьютерах или на одном и том же компьютере.

В этой главе рассматриваются вопросы создания программ на языке Python, которые соединяются с другими процессами с помощью стандартной библиотеки `socket` (а также библиотек, реализованных на основе библиотеки `socket`) и затем взаимодействуют с этими процессами.

Сетевые клиенты

Роль серверов – находиться в ожидании, когда клиенты соединятся с ними, а роль клиентов – инициировать соединения. В состав стандартной библиотеки языка Python входят реализации множества сетевых клиентов. В этом разделе мы обсудим наиболее типичные и часто используемые разновидности клиентов.

socket

Модуль `socket` реализует интерфейс доступа к реализации сокетов операционной системы. Это означает, что на языке Python можно реализовать любые действия с сокетами. На случай, если ранее вам не приходилось заниматься разработкой программ, работающих с сетью, эта глава предоставляет краткий обзор средств работы в сети. Это должно помочь вам составить представление о том, какие действия можно реализовать с помощью сетевых библиотек языка Python.

Модуль socket содержит фабричную функцию socket(), которая создает и возвращает объект socket. Несмотря на то, что функция socket() может принимать большое число аргументов, определяющих тип создаваемого сокета, при вызове функции socket() без аргументов по умолчанию возвращается объект сокет TCP/IP:

```
In [1]: import socket

In [2]: s = socket.socket()

In [3]: s.connect(('192.168.1.15', 80))

In [4]: s.send("GET / HTTP/1.0\n\n")
Out[4]: 16

In [5]: s.recv(200)
Out[5]: 'HTTP/1.1 200 OK\r\n\
Date: Mon, 03 Sep 2007 18:25:45 GMT\r\n\
Server: Apache/2.0.55 (Ubuntu) DAV/2 PHP/5.1.6\r\n\
Content-Length: 691\r\n\
Connection: close\r\n\
Content-Type: text/html; charset=UTF-8\r\n\
\r\n\
<!DOCTYPE HTML P'
In [6]: s.close()
```

В этом примере с помощью фабричной функции socket() создается объект типа socket с именем s. После этого он подключается к порту 80 (номер порта, используемый протоколом HTTP по умолчанию) локального веб-сервера. Затем серверу передается текстовая строка "GET / HTTP/1.0\n\n" (которая представляет собой простейший запрос HTTP). После посылки запроса сокет получает первые 200 байтов ответа сервера, в котором содержится сообщение о состоянии "200 OK" и заголовки HTTP. В самом конце мы закрываем соединение.

В этом примере демонстрируется использование методов объекта socket, к которым вы, вероятно, будете обращаться наиболее часто. Метод connect() устанавливает соединение между вашим объектом socket и удаленным сокетом (то есть «не с этим объектом сокета»). Метод send() выполняет передачу данных от вашего объекта socket удаленному хосту. Метод recv() выполняет прием любых данных, которые были переданы удаленным хостом. И метод close() закрывает соединение между двумя сокетами. Этот очень простой пример демонстрирует, насколько легко можно создавать объекты socket и с их помощью осуществлять передачу и прием данных.

Теперь рассмотрим немного более полезный пример. Предположим, что у вас имеется сервер, на котором выполняется сетевое приложение, такое как веб-сервер. И вам необходимо следить за его работой, чтобы гарантировать возможность соединения с ним в течение дня. Это очень простой вид мониторинга, но он позволяет убедиться, что

сервер продолжает работу и что веб-сервер по-прежнему ожидает соединений с клиентами на некотором порту. Взгляните на пример 5.1.

Пример 5.1. Проверка порта TCP

```
#!/usr/bin/env python

import socket
import re
import sys

def check_server(address, port):
    #Создать сокет TCP
    s = socket.socket()
    print "Attempting to connect to %s on port %s" % (address, port)
    try:
        s.connect((address, port))
        print "Connected to %s on port %s" % (address, port)
        return True
    except socket.error, e:
        print "Connection to %s on port %s failed: %s" % (address, port, e)
        return False

if __name__ == '__main__':
    from optparse import OptionParser
    parser = OptionParser()

    parser.add_option("-a", "--address", dest="address", default='localhost',
                      help="ADDRESS for server", metavar="ADDRESS")

    parser.add_option("-p", "--port", dest="port", type="int", default=80,
                      help="PORT for server", metavar="PORT")

    (options, args) = parser.parse_args()
    print 'options: %s, args: %s' % (options, args)
    check = check_server(options.address, options.port)
    print 'check_server returned %s' % check
    sys.exit(not check)
```

Все необходимые действия выполняются функцией `check_server()`. Она создает объект `socket`. Затем пытается установить соединение с указанным номером порта по указанному адресу. Если соединение удалось установить, функция возвращает значение `True`. В случае неудачи метод `socket.connect()` возбуждает исключение, которое перехватывается функцией, и в этом случае она возвращает значение `False`. В разделе `main` этого сценария выполняется вызов функции `check_server()`. В этом разделе выполняется разбор аргументов командной строки, полученных от пользователя, и переданные аргументы преобразуются в соответствующий формат для последующей передачи функции `check_server()`. В процессе своей работы этот сценарий постоянно выводит сообщения о ходе выполнения. Самое последнее, что выводит сценарий, – это возвращаемое значение функции `check_server()`. В качестве собст-

венного кода завершения сценарий возвращает командной оболочке значение, противоположное возвращаемому значению функции `check_server()`. Сделано это для того, чтобы превратить сценарий в более или менее полезную утилиту. Обычно утилиты, подобные этой, возвращают командной оболочке значение `0` в случае успеха и некоторое другое значение, отличное от `0` (обычно некоторое положительное число), в случае неудачи. Ниже приводится пример вывода сценария, полученного при успешном соединении с веб-сервером, к которому мы уже подключались ранее:

```
jmjones@dinkgutsy:code$ python port_checker_tcp.py -a 192.168.1.15 -p 80
options: {'port': 80, 'address': '192.168.1.15'}, args: []
Attempting to connect to 192.168.1.15 on port 80
Connected to 192.168.1.15 on port 80
check_server returned True
```

Последняя строка в выводе, содержащая текст `check_server returned True`, означает, что соединение было благополучно установлено.

Ниже приводится пример вывода сценария, полученного в результате неудачной попытки соединения:

```
jmjones@dinkgutsy:code$ python port_checker_tcp.py -a 192.168.1.15 -p 81
options: {'port': 81, 'address': '192.168.1.15'}, args: []
Attempting to connect to 192.168.1.15 on port 81
Connection to 192.168.1.15 on port 81 failed: (111, 'Connection refused')
check_server returned False
```

Последняя строка в выводе, содержащая текст `check_server returned False`, означает, что попытка соединения не удалась. В предпоследней строке вывода, содержащей текст `Connection to 192.168.1.15 on port 81 failed`, можно увидеть причину: `'Connection refused'` (Соединение отвергнуто). Об этом можно строить самые разнообразные предположения – например, возможно, что на данном сервере нет никаких процессов, обрабатывающих порт с номером 81.

Мы создали три примера, чтобы продемонстрировать, как можно использовать эту утилиту в сценариях на языке командной оболочки. Первый пример представляет собой команду, которая запускает сценарий и выводит слово `SUCCESS` (успешно) в случае успеха. Здесь был использован оператор `&&`, играющий роль условной инструкции if-then:

```
$ python port_checker_tcp.py -a 192.168.1.15 -p 80 && echo "SUCCESS"
options: {'port': 80, 'address': '192.168.1.15'}, args: []
Attempting to connect to 192.168.1.15 on port 80
Connected to 192.168.1.15 on port 80
check_server returned True
SUCCESS
```

В этом случае сценарий благополучно установил соединение, поэтому после того, как он выполнился и вывел результаты, командная оболочка напечатала слово `SUCCES`.

```
$ python port_checker_tcp.py -a 192.168.1.15 -p 81 && echo "FAILURE"
options: {'port': 81, 'address': '192.168.1.15'}, args: []
Attempting to connect to 192.168.1.15 on port 81
Connection to 192.168.1.15 on port 81 failed: (111, 'Connection refused')
check_server returned False
```

На этот раз сценарий завершился неудачей, но, несмотря на это, команда не вывела слово FAILURE (неудача).

```
$ python port_checker_tcp.py -a 192.168.1.15 -p 81 || echo "FAILURE"
options: {'port': 81, 'address': '192.168.1.15'}, args: []
Attempting to connect to 192.168.1.15 on port 81
Connection to 192.168.1.15 on port 81 failed: (111, 'Connection refused')
check_server returned False
FAILURE
```

В этом случае сценарий тоже потерпел неудачу, но на этот раз мы заменили оператор && оператором ||. Это просто означает, что в случае, если сценарий возвращает признак неудачи, следует вывести слово FAILURE, что и было сделано.

Сам факт, что сервер позволяет выполнить подключение к порту с номером 80, еще не означает доступность веб-сервера. Более точно определить состояние веб-сервера поможет тест, который определяет, способен ли веб-сервер генерировать заголовки HTTP с ожидаемым кодом состояния для заданного URL. Сценарий в примере 5.2 реализует именно такой тест.

Пример 5.2. Проверка веб-сервера с помощью сокетов

```
#!/usr/bin/env python

import socket
import re
import sys

def check_webserver(address, port, resource):
    #Создать строку запроса HTTP
    if not resource.startswith('/'):
        resource = '/' + resource
    request_string = "GET %s HTTP/1.1\r\nHost: %s\r\n\r\n" % (resource,
                                                              address)
    print 'HTTP request:'
    print '|||%s|||' % request_string

    #Создать сокет TCP
    s = socket.socket()
    print "Attempting to connect to %s on port %s" % (address, port)
    try:
        s.connect((address, port))
        print "Connected to %s on port %s" % (address, port)
        s.send(request_string)
        #Нам достаточно получить только первые 100 байтов
        rsp = s.recv(100)
```

```
            print 'Received 100 bytes of HTTP response'
            print '|||%s|||' % rsp
        except socket.error, e:
            print "Connection to %s on port %s failed: %s" % (address, port, e)
            return False
        finally:
            #Будучи добропорядочными программистами закроем соединение
            print "Closing the connection"
            s.close()
        lines = rsp.splitlines()
        print 'First line of HTTP response: %s' % lines[0]
        try:
            version, status, message = re.split(r'\s+', lines[0], 2)
            print 'Version: %s, Status: %s, Message: %s' % (version,
                                                            status, message)
        except ValueError:
            print 'Failed to split status line'
            return False
        if status in ['200', '301']:
            print 'Success - status was %s' % status
            return True
        else:
            print 'Status was %s' % status
            return False

    if __name__ == '__main__':
        from optparse import OptionParser
        parser = OptionParser()
        parser.add_option("-a", "--address", dest="address", default='localhost',
                          help="ADDRESS for webserver", metavar="ADDRESS")

        parser.add_option("-p", "--port", dest="port", type="int", default=80,
                          help="PORT for webserver", metavar="PORT")

        parser.add_option("-r", "--resource", dest="resource",
                          default='index.html',
                          help="RESOURCE to check", metavar="RESOURCE")

        (options, args) = parser.parse_args()
        print 'options: %s, args: %s' % (options, args)
        check = check_webserver(options.address, options.port, options.resource)
        print 'check_webserver returned %s' % check
        sys.exit(not check)
```

Как и в предыдущем примере, где основную работу выполняла функция `check_server()`, в этом примере также все необходимые действия выполняются единственной функцией `check_webserver()`. Сначала функция `check_webserver()` создает строку запроса HTTP. Протокол HTTP, если вы не в курсе, достаточно четко регламентирует порядок взаимодействий серверов и клиентов. Запрос HTTP, который создается в функции `check_webserver()`, является чуть ли не самым простым запросом. Затем функция `check_webserver()` создает объект `socket`, с его помощью

устанавливает соединение с сервером и отправляет запрос HTTP. После этого она читает ответ сервера и закрывает соединение. Когда в сокете возникает ошибка, функция возвращает значение `False`, указывая тем самым, что проверка веб-сервера потерпела неудачу. Затем она берет ответ, полученный от сервера, и извлекает из него код состояния. Если код состояния имеет значение 200, что означает «OK» (все в порядке), или 301, что означает «Moved Permanently» (запрошенная страница была перемещена), функция `check_webserver()` возвращает значение `True`, в противном случае она возвращает значение `False`. В разделе `main` сценарий выполняет разбор аргументов командной строки, полученных от пользователя, и вызывает функцию `check_webserver()`. После выполнения функции `check_webserver()` сценарий возвращает командной оболочке значение, противоположное значению, полученному от функции. Сделано это по тем же причинам, что и в предыдущем примере. Нам хотелось бы иметь возможность вызывать этот сценарий из сценариев командной оболочки и определять случаи успеха или неудачи. Ниже приводится пример использования этого сценария:

```
$ python web_server_checker_tcp.py -a 192.168.1.15 -p 80 -r apache2-default
options: {'resource': 'apache2-default', 'port': 80, 'address':
'192.168.1.15'}, args: []
HTTP request:
|||GET /apache2-default HTTP/1.1
Host: 192.168.1.15

|||
Attempting to connect to 192.168.1.15 on port 80
Connected to 192.168.1.15 on port 80
Received 100 bytes of HTTP response
|||HTTP/1.1 301 Moved Permanently
Date: Wed, 16 Apr 2008 23:31:24 GMT
Server: Apache/2.0.55 (Ubuntu) |||
Closing the connection
First line of HTTP response: HTTP/1.1 301 Moved Permanently
Version: HTTP/1.1, Status: 301, Message: Moved Permanently
Success - status was 301
check_webserver returned True
```

Последние четыре строки вывода показывают, что код состояния HTTP для адреса `/apache2-default` на этом сервере имеет значение 301, что означает успех.

Ниже приводится пример повторной попытки. На этот раз мы указали ресурс, отсутствующий на веб-сервере, чтобы продемонстрировать, что произойдет, когда функция `check_webserver()` вернет значение `False`:

```
$ python web_server_checker_tcp.py -a 192.168.1.15 -p 80 -r foo
options: {'resource': 'foo', 'port': 80, 'address': '192.168.1.15'}, args: []
HTTP request:
|||GET /foo HTTP/1.1
Host: 192.168.1.15
```

```
|||
Attempting to connect to 192.168.1.15 on port 80
Connected to 192.168.1.15 on port 80
Received 100 bytes of HTTP response
|||HTTP/1.1 404 Not Found
Date: Wed, 16 Apr 2008 23:58:55 GMT
Server: Apache/2.0.55 (Ubuntu) DAV/2 PH|||
Closing the connection
First line of HTTP response: HTTP/1.1 404 Not Found
Version: HTTP/1.1, Status: 404, Message: Not Found
Status was 404
check_webserver returned False
```

Последние четыре строки в предыдущем примере свидетельствовали об успешной проверке, но в данном случае четыре последние строки вывода показывают, что проверка завершилась неудачей. На этом сервере отсутствует ресурс с адресом `/foo`, поэтому функция проверки вернула значение `False`.

В этом разделе было показано, как создавать низкоуровневые утилиты, устанавливающие соединение с серверами в сети и выполняющие простейшие проверки. Цель этих примеров состояла в том, чтобы показать вам, как серверы и клиенты взаимодействуют друг с другом. Если у вас имеется возможность написать сетевой программный компонент, использующий более высокоуровневую библиотеку, чем модуль `socket`, вам следует использовать ее. Нет смысла тратить свое время на создание сетевых программных компонентов, использующих такую низкоуровневую библиотеку, как модуль `socket`.

httplib

Предыдущий пример продемонстрировал, как можно выполнить запрос HTTP с помощью модуля `socket`. В этом примере будет показано, как то же самое можно реализовать с помощью модуля `httplib`. Как определить, когда предпочтительнее использовать модуль `httplib`, а когда модуль `socket`? Или, в более широком смысле, когда предпочтительнее использовать более высокоуровневый модуль, а когда – более низкоуровневый? Одно хорошее правило, выработанное на практике, гласит – всякий раз, когда имеется такая возможность, следует использовать более высокоуровневый модуль. Возможно, вам потребуется нечто, что пока отсутствует в библиотеке, или вам необходимо организовать более точное управление чем-то, что уже имеется в библиотеке, или вы захотите воспользоваться преимуществами производительности. Но даже в этом случае нет никаких причин полностью отказываться от использования такой библиотеки, как `httplib`, и отдавать предпочтение такой низкоуровневой библиотеке, как `socket`.

Пример 5.3 реализует ту же самую функциональность, что и предыдущий, используя для этого возможности модуля `httplib`.

Пример 5.3. Проверка веб-сервера с помощью httplib

```
#!/usr/bin/env python

import httplib
import sys

def check_webserver(address, port, resource):
    #Создать соединение
    if not resource.startswith('/'):
        resource = '/' + resource
    try:
        conn = httplib.HTTPConnection(address, port)
        print 'HTTP connection created successfully'
        #Выполнить запрос
        req = conn.request('GET', resource)
        print 'request for %s successful' % resource
        #Получить ответ
        response = conn.getresponse()
        print 'response status: %s' % response.status
    except sock.error, e:
        print 'HTTP connection failed: %s' % e
        return False
    finally:
        conn.close()
        print 'HTTP connection closed successfully'
    if response.status in [200, 301]:
        return True
    else:
        return False

if __name__ == '__main__':
    from optparse import OptionParser
    parser = OptionParser()
    parser.add_option("-a", "--address", dest="address", default='localhost',
                      help="ADDRESS for webserver", metavar="ADDRESS")

    parser.add_option("-p", "--port", dest="port", type="int", default=80,
                      help="PORT for webserver", metavar="PORT")

    parser.add_option("-r", "--resource", dest="resource",
                      default='index.html',
                      help="RESOURCE to check", metavar="RESOURCE")

    (options, args) = parser.parse_args()
    print 'options: %s, args: %s' % (options, args)
    check = check_webserver(options.address, options.port, options.resource)
    print 'check_webserver returned %s' % check
    sys.exit(not check)
```

Концептуально этот пример достаточно близок к предыдущему. Два самых существенных отличия состоят в том, что здесь отсутствует необходимость вручную создавать строку запроса HTTP и нет никакой необходимости вручную выполнять анализ полученного ответа. Объ-

ект соединения библиотеки httplib имеет метод request(), который конструирует и отправляет строку запроса HTTP. Кроме того, у объекта соединения имеется метод getresponse(), который создает объект с полученным ответом. Объект ответа предоставляет возможность получить код состояния ответа HTTP, обратившись к атрибуту status. Хотя объем программного кода в этом примере уменьшился ненамного, тем не менее, нам удалось избавиться от необходимости вручную создавать, отправлять и получать запрос и ответ HTTP. Да и выглядит этот пример более аккуратным.

Ниже приводится результат запуска сценария с теми же аргументами командной строки, которые в предыдущем примере привели к успеху. Мы запрашиваем ресурс с адресом / на веб-сервере и находим его:

```
$ python web_server_checker_httplib.py -a 192.168.1.15 -r /
options: {'resource': '/', 'port': 80, 'address': '192.168.1.15'}, args: []
HTTP connection created successfully
request for / successful
response status: 200
HTTP connection closed successfully
check_webserver returned True
```

А ниже – результат запуска сценария с аргументами командной строки, которые в предыдущем примере привели к неудаче. Мы запрашиваем ресурс с адресом /foo на веб-сервере и не находим его:

```
$ python web_server_checker_httplib.py -a 192.168.1.15 -r /foo
options: {'resource': '/foo', 'port': 80, 'address': '192.168.1.15'}, args: []
HTTP connection created successfully
request for /foo successful
response status: 404
HTTP connection closed successfully
check_webserver returned False
```

Как уже говорилось выше, всякий раз, когда имеется возможность использовать более высокоуровневую библиотеку, следует использовать ее. Использование httplib вместо модуля socket оказалось более простым и более ясным. А чем проще программный код, тем ниже вероятность появления ошибок в нем.

ftplib

Помимо модулей httplib и socket в состав стандартной библиотеки языка Python входит также реализация клиента FTP в виде модуля ftplib. Модуль ftplib – это полнофункциональная клиентская библиотека для работы с протоколом FTP, которая позволяет реализовать любые задачи, выполняемые приложениями FTP-клиентов. Например, с ее помощью в сценарии на языке Python можно выполнить вход на FTP-сервер, получить список файлов в определенном каталоге, загрузить файлы, выгрузить файлы, перейти в другой каталог и выйти.

Можно даже воспользоваться одной из множества платформ, предназначенных для реализации графического интерфейса, доступных в языке Python, и создать для работы с протоколом FTP свое приложение с графическим интерфейсом.

Вместо того чтобы дать краткий обзор библиотеки, мы приведем пример 5.4 и затем поясним, как он работает.

Пример 5.4. Загрузка файлов с помощью ftplib

```
#!/usr/bin/env python

from ftplib import FTP
import ftplib
import sys
from optparse import OptionParser

parser = OptionParser()

parser.add_option("-a", "--remote_host_address", dest="remote_host_address",
    help="REMOTE FTP HOST.",
    metavar="REMOTE FTP HOST")

parser.add_option("-r", "--remote_file", dest="remote_file",
    help="REMOTE FILE NAME to download.",
    metavar="REMOTE FILE NAME")

parser.add_option("-l", "--local_file", dest="local_file",
    help="LOCAL FILE NAME to save remote file to", metavar="LOCAL FILE NAME")

parser.add_option("-u", "--username", dest="username",
    help="USERNAME for ftp server", metavar="USERNAME")

parser.add_option("-p", "--password", dest="password",
    help="PASSWORD for ftp server", metavar="PASSWORD")

(options, args) = parser.parse_args()

if not (options.remote_file and
        options.local_file and
        options.remote_host_address):
    parser.error('REMOTE HOST, LOCAL FILE NAME, ' \
            'and REMOTE FILE NAME are mandatory')

if options.username and not options.password:
    parser.error('PASSWORD is mandatory if USERNAME is present')

ftp = FTP(options.remote_host_address)
if options.username:
    try:
        ftp.login(options.username, options.password)
    except ftplib.error_perm, e:
        print "Login failed: %s" % e
        sys.exit(1)
else:
    try:
```

```
            ftp.login()
        except ftplib.error_perm, e:
            print "Anonymous login failed: %s" % e
            sys.exit(1)
    try:
        local_file = open(options.local_file, 'wb')
        ftp.retrbinary('RETR %s' % options.remote_file, local_file.write)
    finally:
        local_file.close()
        ftp.close()
```

В первой части сценария (сразу вслед за инструкциями, выполняющими разбор аргументов командной строки) создается объект `FTP` вызовом конструктора `FTP()`, которому передается адрес сервера. Можно было бы создать объект FTP, вызвав конструктор без аргументов, и затем вызвать метод `connect()`, передав ему адрес сервера FTP. Затем сценарий выполняет вход на сервер, используя имя пользователя и пароль, если таковые были указаны, в противном случае выполняется анонимный вход. Далее он создает объект типа `file`, куда будут сохраняться данные, получаемые из файла на сервере FTP. После этого вызывается метод `retrbinary()` объекта `FTP`. Метод `retrbinary()`, как следует из его имени, получает двоичный файл с сервера FTP. Метод принимает два параметра: команду, извлекающую файл, и функцию обратного вызова. Обратите внимание, что в качестве функции обратного вызова используется метод `write()` объекта `file`, который был создан на предыдущем шаге. Важно отметить, что в этом случае мы сами не вызываем метод `write()`. Мы передаем метод `write()` методу `retrbinary()`, чтобы он сам мог вызывать метод `write()`. Метод `retrbinary()` будет вызывать функцию обратно вызова при получении каждого блока данных, получаемого от сервера FTP. Функция обратного вызова может выполнять над данными любые действия. Например, эта функция могла бы просто подсчитывать число байтов, принимаемых от сервера FTP. Передача метода `write()` объекта `file` приводит к тому, что данные, получаемые от сервера FTP, будут записаны в объект `file`. В заключение сценарий закрывает объект `file` и соединение с сервером FTP. В сценарии предусматривается обработка ошибок: процедура получения двоичного файла с сервера FTP заключена в инструкцию `try`, а в блоке `finally` выполняется закрытие локального файла и соединения с сервером FTP. Если случится что-то непредвиденное, файл и соединение будут закрыты перед завершением сценария. Краткое обсуждение функций обратного вызова вы найдете в приложении.

urllib

Перемещаясь ко все более высокоуровневым модулям, входящим в состав стандартной библиотеки, мы наконец достигли модуля `urllib`. Часто, рассматривая возможность применения библиотеки `urllib`, предполагают использовать ее для работы с протоколом HTTP, забывая, что

ресурсы FTP также можно идентифицировать посредством URL. Поэтому вы, возможно, даже не предполагали, что библиотека `urllib` позволяет обращаться к ресурсам FTP, хотя такая возможность существует. Пример 5.5 обладает той же функциональностью, что и предыдущий пример, созданный на основе использования `ftplib`, но использует модуль `urllib`.

Пример 5.5. Загрузка файлов с помощью urllib

```
#!/usr/bin/env python
"""
url retriever

Порядок использования:

url_retrieve_urllib.py URL FILENAME

URL:
Если адрес URL - это адрес FTP URL, то формат адреса должен быть следующим:
ftp://[username[:password]@]hostname/filename
Если имеется необходимость использовать абсолютный путь к загружаемому файлу,
Вы должны определить URL, который выглядит примерно так:
ftp://user:password@host/%2Fpath/to/myfile.txt
Обратите внимание на последовательность '%2F' в начале пути к файлу.

FILENAME:
Абсолютный или относительный путь к локальному файлу
"""

import urllib
import sys

if '-h' in sys.argv or '--help' in sys.argv:
    print __doc__
    sys.exit(1)

if not len(sys.argv) == 3:
    print 'URL and FILENAME are mandatory'
    print __doc__
    sys.exit(1)
url = sys.argv[1]
filename = sys.argv[2]
urllib.urlretrieve(url, filename)
```

Этот сценарий выглядит короче и опрятнее. Он наглядно демонстрирует мощь библиотеки `urllib`. На самом деле, большую часть сценария занимает документация, описывающая порядок его использования. Более того, даже для анализа аргументов командной строки потребовалось больше программного кода, чем для фактического выполнения действий. Мы решили упростить процедуру анализа аргументов командной строки. Поскольку оба аргумента являются обязательными, мы предпочли использовать позиционные аргументы и избавиться от ключей. По сути, всю работу в этом примере выполняет единственная строка:

```
urllib.urlretrieve(url, filename)
```

После получения аргументов командной строки с помощью `sys.argv` эта строка обращается по указанному адресу URL и сохраняет полученные данные в локальном файле с указанным именем. Этот сценарий будет работать как с адресами HTTP, так и с адресами FTP, и будет работать, даже когда имя пользователя и пароль включены в URL.

Ценность этого примера заключается в следующем: если вы предположили, что в языке Python некоторые действия должны выполняться проще, чем в других языках программирования, скорее всего так оно и окажется. Наверняка имеется какая-нибудь высокоуровневая библиотека, которая реализует именно то, что вам необходимо, и эта библиотека входит в состав стандартной библиотеки языка Python. В данном случае библиотека `urllib` реализует именно то, что необходимо, и оказалось достаточно заглянуть в документацию к стандартной библиотеке, чтобы узнать об этом. Впрочем, иногда вам, возможно, придется покидать стандартную библиотеку и искать другие ресурсы Python, такие как каталог пакетов Python (Python Package Index, PyPI) по адресу: *http://pypi.python.org/pypi*.

urllib2

Примером другой высокоуровневой библиотеки является библиотека `urllib2`. Эта библиотека реализует практически те же самые функциональные возможности, что и библиотека `urllib`. Например, `urllib2` обладает улучшенной поддержкой аутентификации и улучшенной поддержкой cookie. Поэтому, когда вы начнете использовать библиотеку `urllib` и обнаружите, что вам чего-то не хватает, обратитесь к библиотеке `urllib2` – возможно, она лучше будет соответствовать вашим потребностям.

Средства вызова удаленных процедур

Как правило, причиной создания сценариев для работы с сетью становится необходимость организации взаимодействий между процессами. Часто бывает вполне достаточно ограничиться простыми взаимодействиями, например, с помощью протокола HTTP или сокетов. Однако, иногда возникает необходимость выполнять программный код в разных процессах и даже на разных компьютерах, как если бы это был один и тот же процесс. Если бы у вас имелась возможность выполнять программный код удаленно, в некотором другом процессе, запущенном из программы на языке Python, то вы, скорее всего, хотели бы, чтобы возвращаемые значения таких удаленных вызовов были объектами языка Python, работать с которыми намного проще, чем с фрагментами текста, которые необходимо анализировать вручную. Так вот, существует несколько инструментов, позволяющих организовать вызов удаленных процедур (Remote Procedure Call, RPC).

XML–RPC

Технология XML-RPC, позволяющая организовать вызов удаленных процедур, основана на обмене специально сформированными документами XML между двумя процессами. Однако пусть вас не беспокоит часть XML в названии – вам, скорее всего, даже не придется вникать в формат документов, которыми будут обмениваться процессы. Единственное, что вам действительно необходимо знать, чтобы использовать технологию XML-RPC – это то, что в стандартной библиотеке языка Python имеются реализации как клиентской, так и серверной частей этой технологии. К тому же, вам полезно будет узнать, что реализации XML-RPC имеются в большинстве языков программирования и что эта технология очень проста в использовании.

В примере 5.6 приводится реализация простого сервера XML-RPC.

Пример 5.6. Простой сервер XML-RPC

```
#!/usr/bin/env python

import SimpleXMLRPCServer
import os

def ls(directory):
    try:
        return os.listdir(directory)
    except OSError:
        return []

def ls_boom(directory):
    return os.listdir(directory)

def cb(obj):
    print "OBJECT::", obj
    print "OBJECT.__class__::", obj.__class__
    return obj.cb()

if __name__ == '__main__':
    s = SimpleXMLRPCServer.SimpleXMLRPCServer(('127.0.0.1', 8765))
    s.register_function(ls)
    s.register_function(ls_boom)
    s.register_function(cb)
    s.serve_forever()
```

Этот сценарий создает новый объект `SimpleXMLRPCServer` и связывает его с портом 8765 и с петлевым интерфейсом, имеющим IP-адрес 127.0.0.1, что делает его доступным только для процессов, выполняющихся на данном компьютере. Затем сценарий регистрирует функции `ls()` и `ls_boom()`, которые определены тут же, в сценарии. Назначение функции `cb()` мы объясним чуть погодя. Функция `ls()` с помощью `os.listdir()` получает содержимое указанного каталога и возвращает его в виде списка. Функция `ls()` маскирует любые исключения `OSError`, которые только могут возникнуть. Функция `ls_boom()` не выполняет обработку

исключений и возвращает их клиенту XML-RPC. После этого сценарий входит в бесконечный цикл `serve_forever()`, в котором он ожидает поступления запросов от клиентов и обрабатывает их. Ниже приводится пример взаимодействия с этим сервером в оболочке IPython:

```
In [1]: import xmlrpclib

In [2]: x = xmlrpclib.ServerProxy('http://localhost:8765')

In [3]: x.ls('.')
Out[3]:
['.svn',
 'web_server_checker_httplib.py',
....
 'subprocess_arp.py',
 'web_server_checker_tcp.py']

In [4]: x.ls_boom('.')
Out[4]:
['.svn',
 'web_server_checker_httplib.py',
....
 'subprocess_arp.py',
 'web_server_checker_tcp.py']

In [5]: x.ls('/foo')
Out[5]: []

In [6]: x.ls_boom('/foo')
---------------------------------------------------------------------------
<class 'xmlrpclib.Fault'>                 Traceback (most recent call last)
...
.
.
<<большой блок диагностической информации>>
.
.
...
    786 if self._type == "fault":
--> 787 raise Fault(**self._stack[0])
    788 return tuple(self._stack)
    789

<class 'xmlrpclib.Fault'>: <Fault 1: "<type 'exceptions.OSError'>
    :[Errno 2] No such file or directory: '/foo'">
   (:[Errno 2] Нет такого файла или каталога: '/foo')
```

Прежде всего мы создали объект `ServerProxy`, указав ему адрес сервера XML-RPC. Затем мы вызвали функцию `x.ls('.')`, чтобы получить содержимое текущего рабочего каталога. Сервер был запущен из каталога, содержащего программный код примеров к этой книге, поэтому список включает файлы примеров. Самое интересное, что на стороне клиента функция `x.ls('.')` возвращает список языка Python. Независимо

от языка реализации сервера – Java, Perl, Ruby или C# – можно рассчитывать на получение подобного результата. На языке реализации сервера может быть получен перечень файлов в каталоге, создан список, массив или коллекция имен файлов; после этого программный код сервера XML-RPC может преобразовать этот список или массив в формат XML и передать его обратно клиенту. Мы также попробовали вызвать функцию `ls_boom()`. Благодаря тому, что в функции `ls_boom()` не предусматривается обработка исключений, в отличие от `ls()`, мы смогли увидеть, как исключение передается от сервера клиенту. Более того, на стороне клиента мы увидели даже диагностическую информацию.

Возможности функциональной совместимости, которыми обладает технология XML-RPC, безусловно интересны. Но гораздо более интересен тот факт, что существует возможность написать некоторый программный код, запустить его на произвольном числе машин и затем вызывать этот код удаленно в случае необходимости.

Однако в технологии XML-RPC имеются свои ограничения. Эти ограничения могут представлять определенную проблему, или сама технология может не соответствовать нуждам и чаяниям разработчика. Например, когда удаленному программному коду передается нестандартный объект на языке Python, библиотека XML-RPC преобразует этот объект в словарь, переводит его в формат XML и отправляет удаленной стороне. Безусловно, вы сможете обработать эту ситуацию, но для этого потребуется написать программный код, который будет извлекать данные из XML-версии словаря, чтобы превратить его обратно в оригинальный объект. Так почему бы не использовать объекты непосредственно на сервере RPC, чтобы избежать таких сложностей с преобразованиями? Это нельзя сделать с помощью XML-RPC, но существуют другие возможности.

Pyro

Pyro – это платформа, которая лишена некоторых недостатков, свойственных XML-RPC. Название Pyro происходит от Python Remote Objects (удаленные объекты Python). Она позволяет реализовать те же самые действия, которые позволяет XML-RPC, но вместо того, чтобы преобразовывать объекты в форму словаря, она обеспечивает возможность передачи информации о типе вместе с самим объектом. Если вы действительно захотите воспользоваться платформой Pyro, вам придется установить ее отдельно. Она не поставляется вместе с Python. Кроме того, вы должны понимать, что Pyro работает только со сценариями на языке Python, тогда как технология XML-RPC в состоянии обеспечить взаимодействие между сценариями на языке Python и программами, написанными на других языках. В примере 5.7 приводится реализация той же самой функции `ls()`, что и в примере, использующем технологию XML-RPC.

Пример 5.7. Простой сервер Pyro

```
#!/usr/bin/env python

import Pyro.core
import os
from xmlrpc_pyro_diff import PSACB

class PSAExample(Pyro.core.ObjBase):

    def ls(self, directory):
        try:
            return os.listdir(directory)
        except OSError:
            return []

    def ls_boom(self, directory):
        return os.listdir(directory)

    def cb(self, obj):
        print "OBJECT:", obj
        print "OBJECT.__class__:", obj.__class__
        return obj.cb()

if __name__ == '__main__':
    Pyro.core.initServer()
    daemon=Pyro.core.Daemon()
    uri=daemon.connect(PSAExample(),"psaexample")

    print "The daemon runs on port:",daemon.port
    print "The object's uri is:",uri
    daemon.requestLoop()
```

Пример на базе Pyro похож на пример XML-RPC. Сначала мы создали класс `PSAExample` **с методами** `ls()`**,** `ls_boom()` **и** `cb()`**. Затем из глубин Pyro был извлечен демон. После этого мы связали объект** `PSAExample` **с демоном. Наконец, мы запустили демон для обслуживания запросов.**

Ниже приводится сеанс взаимодействия с сервером Pyro в оболочке IPython:

```
In [1]: import Pyro.core
/usr/lib/python2.5/site-packages/Pyro/core.py:11: DeprecationWarning:
The sre module is deprecated, please import re.
  import sys, time, sre, os, weakref

In [2]: psa = Pyro.core.getProxyForURI("PYROLOC://localhost:7766/psaexample")
Pyro Client Initialized. Using Pyro V3.5

In [3]: psa.ls(".")
Out[3]:
['pyro_server.py',
....
 'subprocess_arp.py',
 'web_server_checker_tcp.py']

In [4]: psa.ls_boom('.')
```

```
Out[4]:
['pyro_server.py',
....
 'subprocess_arp.py',
 'web_server_checker_tcp.py']

In [5]: psa.ls("/foo")
Out[5]: []

In [6]: psa.ls_boom("/foo")
---------------------------------------------------------------------------
<type 'exceptions.OSError'>                Traceback (most recent call last)

/home/jmjones/local/Projects/psabook/oreilly/<ipython console> in <module>()
.
.
...
<<большой блок диагностической информации>>
...
.
.
--> 115 raise self.excObj
    116 def __str__(self):
    117 s=self.excObj.__class__.__name__

<type 'exceptions.OSError'>: [Errno 2] No such file or directory: '/foo'
(type 'exceptions.OSError'>: [Errno 2] Нет такого файла или каталога: '/foo')
```

Отлично. Мы получили те же результаты, что и в примере XML-RPC. Именно этого мы и ожидали. Но что произойдет, если попробовать передать нестандартный объект? Попробуем определить новый класс, создать из него объект и передать его функции `cb()` в реализации на основе XML-RPC и методу `cb()` в реализации на основе Pyro. В примере 5.8 приводится фрагмент программного кода, который будет выполняться.

Пример 5.8. Различия между XML-RPC и Pyro

```
import Pyro.core
import xmlrpclib

class PSACB:
    def __init__(self):
        self.some_attribute = 1

    def cb(self):
        return "PSA callback"

if __name__ == '__main__':
    cb = PSACB()

    print "PYRO SECTION"
    print "*" * 20
    psapyro = Pyro.core.getProxyForURI("PYROLOC://localhost:7766/psaexample")
    print "-->>", psapyro.cb(cb)
```

```
print "*" * 20

print "XML-RPC SECTION"
print "*" * 20
psaxmlrpc = xmlrpclib.ServerProxy('http://localhost:8765')
print "-->>", psaxmlrpc.cb(cb)
print "*" * 20
```

Обращение к функции `cb()` в обеих реализациях, XML-RPC и Pyro, должно привести к вызову метода `cb()` переданного объекта. И в обоих случаях этот метод должен вернуть строку `PSA callback`. Ниже показано, что произошло, когда мы запустили этот сценарий:

```
jmjones@dinkgutsy:code$ python xmlrpc_pyro_diff.py
/usr/lib/python2.5/site-packages/Pyro/core.py:11: DeprecationWarning:
The sre module is deprecated, please import re.
  import sys, time, sre, os, weakref
PYRO SECTION
********************
Pyro Client Initialized. Using Pyro V3.5
-->> PSA callback
********************
XML-RPC SECTION
********************
-->>
Traceback (most recent call last):
  File "xmlrpc_pyro_diff.py", line 23, in <module>
    print "-->>", psaxmlrpc.cb(cb)
  File "/usr/lib/python2.5/xmlrpclib.py", line 1147, in __call__
    return self.__send(self.__name, args)
  File "/usr/lib/python2.5/xmlrpclib.py", line 1437, in __request
    verbose=self.__verbose
  File "/usr/lib/python2.5/xmlrpclib.py", line 1201, in request
    return self._parse_response(h.getfile(), sock)
  File "/usr/lib/python2.5/xmlrpclib.py", line 1340, in _parse_response
    return u.close()
  File "/usr/lib/python2.5/xmlrpclib.py", line 787, in close
    raise Fault(**self._stack[0])
xmlrpclib.Fault: <Fault 1: "<type 'exceptions.AttributeError'>:'dict' object
    has no attribute 'cb'">
(xmlrpclib.Fault: <Fault 1: "<type 'exceptions.AttributeError'>:'dict' объект
    не имеет атрибут 'cb'">)
```

Реализация на основе Pyro работает, но реализация на основе XML-RPC потерпела неудачу и оставила нас наедине с кучей диагностической информации. Последняя строка в этом блоке информации сообщает, что в объекте `dict` отсутствует атрибут `cb`. Эта строка обретет смысл, если взглянуть на вывод, полученный от сервера XML-RPC. Вспомните, что в функции `cb()` имеется пара инструкций `print`, которые выводят дополнительную информацию о том, что происходит. Ниже приводится вывод сервера XML-RPC:

```
OBJECT:: {'some_attribute': 1}
OBJECT.__class__:: <type 'dict'>
localhost - - [17/Apr/2008 16:39:02] "POST /RPC2 HTTP/1.0" 200 -
```

После преобразования в словарь объекта, который был создан в клиенте, реализованном на базе XML-RPC, атрибут `some_attribute` превратился в ключ словаря. Атрибут сохранился при передаче объекта на сервер, а метод `cb()` был утрачен.

Ниже приводится вывод сервера Pyro:

```
OBJECT: <xmlrpc_pyro_diff.PSACB instance at 0x9595a8>
OBJECT.__class__: xmlrpc_pyro_diff.PSACB
```

Обратите внимание, что класс объекта – `PSACB`, т.е. тот, который и был создан. На стороне сервера на основе Pyro мы должны включить программный код, импортирующий тот же программный код, который используется клиентом. Это означает, что сервер Pyro вынужден импортировать программный код клиента. Для сериализации объектов платформа Pyro использует стандартный модуль `pickle`, что объясняет, почему Pyro обладает схожим поведением.

Подводя итоги, можно сказать: если вам необходимо простое решение RPC, не имеющее внешних зависимостей, если вам не мешают имеющиеся ограничения XML-RPC, и вам важна поддержка функциональной совместимости с другими языками программирования, то, скорее всего, хорошим выбором будет XML-RPC. С другой стороны, если ограничения XML-RPC слишком тесны для вас, вы не возражаете против установки дополнительных библиотек и предполагаете ограничиться только языком Python, то наилучшим вариантом для вас будет Pyro.

SSH

SSH – это невероятно мощный и широко используемый протокол. Его можно также воспринимать и как инструмент, потому что наиболее распространенная его реализация носит то же самое имя. SSH обеспечивает безопасное соединение с удаленным сервером, выполнение команд оболочки, передачу файлов и переназначение портов в обоих направлениях через соединение.

Если у вас имеется утилита командной строки `ssh`, почему бы тогда не воспользоваться протоколом SSH в сценарии? Вся прелесть здесь состоит в том, что вы получаете всю мощь протокола SSH в комбинации с широкими возможностями языка Python.

Протокол SSH2 реализован на языке Python в виде библиотеки с именем `paramiko`. Из сценариев, которые содержат только программный код на языке Python, вы можете организовать подключение к серверу SSH и реализовать выполнение задач SSH. В примере 5.9 демонстрируется, как можно выполнить соединение с сервером SSH и выполнить простую команду.

Пример 5.9. Соединение с сервером SSH и выполнение команды в удаленной системе

```
#!/usr/bin/env python

import paramiko

hostname = '192.168.1.15'
port = 22
username = 'jmjones'
password = 'xxxYYYxxx'

if __name__ == "__main__":
    paramiko.util.log_to_file('paramiko.log')
    s = paramiko.SSHClient()
    s.load_system_host_keys()
    s.connect(hostname, port, username, password)
    stdin, stdout, stderr = s.exec_command('ifconfig')
    print stdout.read()
    s.close()
```

Как видно из листинга, мы импортируем модуль `paramiko` и определяем три переменные. Затем создаем объект `SSHClient`. После этого производится загрузка ключей хоста, которые в операционной системе Linux извлекаются из файла «known_hosts». Затем выполняется соединение с сервером SSH. Ни в одном из этих действий нет ничего сложного, особенно, если вы уже знакомы с SSH.

Теперь мы готовы выполнить команду в удаленной системе. Метод `exec_command()` выполняет указанную команду и возвращает три файловых дескриптора, ассоциированных с выполняемой командой: стандартный ввод, стандартный вывод и стандартный вывод сообщений об ошибках. Чтобы показать, что команда выполняется на машине с IP-адресом, который был использован для создания соединения SSH, мы вывели результаты выполнения команды `ifconfig` на удаленном сервере:

```
jmjones@dinkbuntu:~/code$ python paramiko_exec.py
eth0      Link encap:Ethernet HWaddr XX:XX:XX:XX:XX:XX
          inet addr:192.168.1.15 Bcast:192.168.1.255 Mask:255.255.255.0
          inet6 addr: xx00::000:x0xx:xx0x:0x00/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1
          RX packets:9667336 errors:0 dropped:0 overruns:0 frame:0
          TX packets:11643909 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:1427939179 (1.3 GiB) TX bytes:2940899219 (2.7 GiB)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1 Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING MTU:16436 Metric:1
          RX packets:123571 errors:0 dropped:0 overruns:0 frame:0
          TX packets:123571 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:94585734 (90.2 MiB) TX bytes:94585734 (90.2 MiB)
```

Выглядит так, как если бы мы выполнили команду `ifconfig` на локальной машине, только IP-адрес отличается.

В примере 5.10 показано, как можно с помощью модуля `paramiko` выполнять передачу файлов по протоколу SFTP между удаленной и локальной машинами. В данном случае пример только получает файлы с удаленной машины, используя для этого метод `get()`. Если у вас возникнет потребность передать файл на удаленную машину, вы можете воспользоваться методом `put()`.

Пример 5.10. Получение файлов с сервера SSH

```
#!/usr/bin/env python

import paramiko
import os

hostname = '192.168.1.15'
port = 22
username = 'jmjones'
password = 'xxxYYYxxx'
dir_path = '/home/jmjones/logs'

if __name__ == "__main__":
    t = paramiko.Transport((hostname, port))
    t.connect(username=username, password=password)
    sftp = paramiko.SFTPClient.from_transport(t)
    files = sftp.listdir(dir_path)
    for f in files:
        print 'Retrieving', f
        sftp.get(os.path.join(dir_path, f), f)
    t.close()
```

В случае, если вы захотите использовать открытые/закрытые ключи вместо паролей, в примере 5.11 приводится модифицированная версия сценария, выполняющего команду в удаленной системе, с использованием ключа RSA.

Пример 5.11. Соединение с сервером SSH и удаленное выполнение команды – с использованием закрытого ключа

```
#!/usr/bin/env python

import paramiko

hostname = '192.168.1.15'
port = 22
username = 'jmjones'
pkey_file = '/home/jmjones/.ssh/id_rsa'

if __name__ == "__main__":
    key = paramiko.RSAKey.from_private_key_file(pkey_file)
    s = paramiko.SSHClient()
    s.load_system_host_keys()
    s.connect(hostname, port, pkey=key)
```

```
stdin, stdout, stderr = s.exec_command('ifconfig')
print stdout.read()
s.close()
```

И в примере 5.12 приводится модифицированная версия сценария, выполняющего передачу файлов и использующего ключ RSA.

Пример 5.12. Получение файлов с сервера SSH

```
#!/usr/bin/env python

import paramiko
import os

hostname = '192.168.1.15'
port = 22
username = 'jmjones'
dir_path = '/home/jmjones/logs'
pkey_file = '/home/jmjones/.ssh/id_rsa'

if __name__ == "__main__":
    key = paramiko.RSAKey.from_private_key_file(pkey_file)
    t = paramiko.Transport((hostname, port))
    t.connect(username=username, pkey=key)
    sftp = paramiko.SFTPClient.from_transport(t)
    files = sftp.listdir(dir_path)
    for f in files:
        print 'Retrieving', f
        sftp.get(os.path.join(dir_path, f), f)
    t.close()
```

Twisted

Twisted – это платформа разработки сетевых приложений на языке Python, управляемых событиями. Эта платформа позволяет решать практически любые задачи, имеющие отношение к работе в сети. Как и любое единое комплексное решение, эта платформа отличается высокой сложностью. Twisted начнет становиться понятной только после неоднократной работы с ней, а в самом начале работа с платформой может оказаться непростым делом. Кроме того, изучение Twisted представляет собой настолько большой труд, что поиск отправного пункта в решении конкретной проблемы часто может оказаться устрашающим.

Тем не менее, мы настоятельно рекомендуем познакомиться с этой платформой и оценить, насколько она соответствует вашему образу мыслей. Если вы легко сможете настроиться на «твистовое» мышление, то изучение платформы Twisted скорее всего будет ценным вложением усилий. Отличной отправной точкой в изучении может служить книга «Twisted Network Programming Essentials» Эйба Феттига (Abe Fettig) (O'Reilly). Эта книга поможет уменьшить влияние отрицательных факторов, о которых упоминалось выше.

Twisted – это сетевая платформа, управляемая событиями, то есть вам придется сконцентрироваться не на написании программного кода, который инициализирует созданные соединения и закрывает их, и на низкоуровневых деталях приема данных, а на программном коде, обрабатывающем происходящие события.

Какие преимущества вы получите при использовании платформы Twisted? Эта платформа стимулирует, а иногда даже вынуждает вас разбивать свои задачи на маленькие части. Организация сетевого соединения отделена от логики обработки событий, происходящих после установления соединения. Эти два фактора до некоторой степени обеспечивают автоматическую возможность повторного использования вашего программного кода. Еще одно преимущество, которое несет в себе применение платформы Twisted, заключается в том, что вам не придется беспокоиться о низкоуровневых подключениях и заниматься обработкой ошибок, возникающих в них. Ваша основная задача заключается в том, чтобы решить, что необходимо предпринять при появлении определенных событий.

В примере 5.13 приводится сценарий, проверяющий сетевой порт и реализованный на платформе Twisted. Это очень простой сценарий, но он наглядно демонстрирует управляемую событиями природу Twisted, в чем вы убедитесь при изучении пояснений к программному коду. Но перед этим мы коснемся нескольких основных концепций, которые вам необходимо знать. В число этих концепций входят реакторы, фабрики, протоколы и отложенное выполнение. Реакторы – это основа главного цикла обработки событий любого приложения, основанного на платформе Twisted. Реакторы занимаются доставкой событий, сетевыми взаимодействиями и многопоточным выполнением. Фабрики отвечают за создание новых экземпляров протоколов. Каждый экземп-

Twisted

У большинства из тех, кто пишет программный код, складывается отчетливая аналогия логике выполнения программы или сценария: это похоже на ручей, текущий по склону холма, ныряющий в провалы, ветвящийся и т. п. Такой программный код легко писать и отлаживать. Программный код, использующий особенности платформы Twisted, совершенно иной. Вследствие асинхронной природы его скорее можно сравнить с падающими каплями дождя, чем с ручьем, текущим по склону, но на этом аналогии и заканчиваются. Платформа вводит новый компонент: перехватчик событий (реактор) сотоварищи. Чтобы написать и отладить программный код, использующий Twisted, придется отказаться от привычных убеждений и начать изобретать аналогии под другую логику выполнения.

ляр фабрики может порождать экземпляры только одного типа протокола. Протоколы определяют принцип действия определенного соединения. Во время выполнения приложения для каждого соединения создается свой экземпляр протокола. А механизм отложенного выполнения обеспечивает возможность объединения действий в цепочки.

Пример 5.13. Проверка порта, реализованная на платформе Twisted

```
#!/usr/bin/env python

from twisted.internet import reactor, protocol
import sys

class PortCheckerProtocol(protocol.Protocol):
    def __init__(self):
        print "Created a new protocol"
    def connectionMade(self):
        print "Connection made"
        reactor.stop()

class PortCheckerClientFactory(protocol.ClientFactory):
    protocol = PortCheckerProtocol
    def clientConnectionFailed(self, connector, reason):
        print "Connection failed because", reason
        reactor.stop()

if __name__ == '__main__':
    host, port = sys.argv[1].split(':')
    factory = PortCheckerClientFactory()
    print "Testing %s" % sys.argv[1]
    reactor.connectTCP(host, int(port), factory)
    reactor.run()
```

Обратите внимание, что здесь мы определили два класса (`PortCheckerProtocol` и `PortCheckerClientFactory`), каждый из которых наследует классы платформы Twisted. Мы связали свою фабрику `PortCheckerClientFactory` с `PortCheckerProtocol`, присвоив класс `PortCheckerProtocol` атрибуту `protocol` класса `PortCheckerClientFactory`. Если попытка установить соединение окончится неудачей, будет вызван метод фабрики `clientConnectionFailed()`. Метод `clientConnectionFailed()` является общим для всех фабрик платформы Twisted, и это единственный метод, который мы определили для нашей фабрики. Определяя метод, «поставляемый» вместе с фабричным классом, мы переопределили поведение этого класса, заданное по умолчанию. Когда на стороне клиента попытка установить соединение терпит неудачу, мы выводим соответствующее сообщение и останавливаем работу реактора.

`PortCheckerProtocol` – это представитель протоколов, о которых говорилось выше. Экземпляр этого класса будет создан сразу же после того, как будет установлено соединение с сервером, порт которого проверяет сценарий. В классе `PortCheckerProtocol` мы определили единственный метод: `connectionMade()`. Этот метод является общим для всех классов

протоколов платформы Twisted. Определяя этот метод, мы тем самым переопределяем поведение по умолчанию. Когда соединение будет благополучно установлено, платформа Twisted вызовет метод `connectionMade()` этого протокола. Как видно из сценария, этот метод просто выводит сообщение и останавливает реактор. (К реакторам мы подойдем очень скоро.)

Здесь оба метода – `connectionMade()` и `clientConnectionFailed()` – демонстрируют «управляемую событиями» природу платформы Twisted. Установленное соединение – это событие. Точно так же событием является и неудача при попытке установить соединение. Когда возникают такие события, платформа Twisted вызывает соответствующие методы, выполняющие их обработку, которые так и называются – обработчики событий.

В основном разделе этого сценария мы создаем экземпляр класса `PortCheckerClientFactory`. Затем предписываем реактору платформы Twisted с помощью заданной фабрики выполнить подключение к заданному порту указанного сервера (эти значения передаются сценарию как аргументы командной строки). После того как реактору будет дано указание подключиться к заданному порту указанного сервера, мы запускаем его в работу. Если этого не сделать, тогда вообще ничего не произойдет.

С точки зрения хронологического порядка выполнения можно сказать, что мы запускаем реактор после того, как дадим ему указание. В данном случае было указано установить соединение с портом сервера и использовать класс `PortCheckerClientFactory` для доставки событий. Если попытка соединения с указанным портом заданного хоста потерпит неудачу, цикл обработки событий вызовет метод `clientConnectionFailed()` класса `PortCheckerClientFactory`. Если соединение будет успешно установлено, фабрика создаст экземпляр протокола `PortCheckerProtocol` и вызовет метод `connectionMade()` этого экземпляра. Завершится ли попытка подключения успехом или неудачей, соответствующий обработчик события остановит реактор и программа завершит свою работу.

Это был очень простой пример, но он демонстрирует суть управляемой событиями природы платформы Twisted. Ключевыми концепциями программирования Twisted, которые не были продемонстрированы в этом примере, являются идея отложенного выполнения и функции обратного вызова. Механизм отложенного выполнения берет на себя обязательство выполнить запрошенное действие. А функции обратного вызова обеспечивают способ, дающий возможность определить требуемое действие. Функции, выполняющие отложенные действия, могут объединяться в цепочки и передавать результаты друг другу. Эта особенность платформы Twisted действительно очень сложна для понимания. (Механизм отложенных действий демонстрируется в примере 5.14.)

В примере 5.14 представлен сценарий, использующий брокер перспективы (Perspective Broker) – уникальный механизм вызова удаленных процедур (RPC) в Twisted. Этот пример представляет собой еще одну реализацию сервера «ls», который ранее в этой же главе был реализован с использованием XML-RPC и Pyro. В первую очередь рассмотрим реализацию сервера.

Пример 5.14. Сервер брокера перспективы на платформе Twisted

```
import os
from twisted.spread import pb
from twisted.internet import reactor

class PBDirLister(pb.Root):
    def remote_ls(self, directory):
        try:
            return os.listdir(directory)
        except OSError:
            return []

    def remote_ls_boom(self, directory):
        return os.listdir(directory)

if __name__ == '__main__':
    reactor.listenTCP(9876, pb.PBServerFactory(PBDirLister()))
    reactor.run()
```

В этом примере определяется единственный класс, `PBDirLister`. Это класс брокера перспективы (`PB`), который действует как удаленный объект, когда клиент соединяется с ним. В этом примере данный класс определяет всего два метода: `remote_ls()` и `remote_ls_boom()`. Метод `remote_ls()` – это один из удаленных методов, которые будут вызываться клиентом. Этот метод `remote_ls()` просто возвращает содержимое указанного каталога. Метод `remote_ls_boom()` выполняет те же действия, что и метод `remote_ls()`, за исключением того, что он не предусматривает обработку исключений. В главном разделе примера мы предписываем брокеру перспективы присоединиться к порту с номером 9876 и запустить реактор.

Пример 5.15. Клиент брокера перспективы платформы Twisted

```
#!/usr/bin/python

from twisted.spread import pb
from twisted.internet import reactor

def handle_err(reason):
    print "an error occurred", reason
    reactor.stop()

def call_ls(def_call_obj):
    return def_call_obj.callRemote('ls', '/home/jmjones/logs')

def print_ls(print_result):
```

```
        print print_result
        reactor.stop()

    if __name__ == '__main__':
        factory = pb.PBClientFactory()
        reactor.connectTCP("localhost", 9876, factory)
        d = factory.getRootObject()
        d.addCallback(call_ls)
        d.addCallback(print_ls)
        d.addErrback(handle_err)
        reactor.run()
```

В этом сценарии клиента определяются три функции: `handle_err()`, `call_ls()` и `print_ls()`. Функция `handle_err()` обрабатывает все возникающие ошибки. Функция `call_ls()` инициирует вызов удаленного метода «ls». Функция `print_ls()` выводит результаты вызова удаленного метода «ls». Может показаться немного странным, что одна функция инициирует вызов удаленного метода, а другая выводит результаты этого вызова. Но, так как Twisted является асинхронной платформой, управляемой событиями, такое положение вещей обретает определенный смысл. Сама платформа способствует созданию программного кода, который делит работу на мелкие части.

В основном разделе примера видно, как реактор определяет, когда и какие функции обратного вызова следует вызывать. Сначала мы создаем фабрику клиента брокера перспективы и предписываем реактору выполнить подключение к порту 9876 сервера `localhost`, используя фабрику клиента `PB` для обработки запросов. Затем вызовом метода `factory.getRootObject()` создается заготовка удаленного объекта. Фактически это объект отложенного действия, поэтому мы имеем возможность объединить действия в конвейер, вызвав метод `addCallback()` объекта.

Первой функцией обратного вызова, которую мы добавляем, является функция `call_ls()`. Функция `call_ls()` вызывает метод `remote_ls()` объекта отложенного действия, созданного на предыдущем шаге. Метод `callRemote()` возвращает сам объект. Вторая функция обратного вызова в цепочке обработки – это функция `print_ls()`. Когда реактор вызывает `print_ls()`, она выводит результаты обращения к удаленному методу `remote_ls()` на предыдущем шаге. Фактически реактор передает результаты вызова удаленного метода функции `print_ls()`. Третья функция обратного вызова в цепочке – `handle_err()`, которая является обычным обработчиком ошибок, она просто сообщает о появлении ошибок в процессе работы. Когда в ходе выполнения возникает ошибка или когда процесс достигает функции `print_ls()`, соответствующие методы останавливают реактор.

Результат работы клиентского сценария выглядит, как показано ниже:

```
jmjones@dinkgutsy:code$ python twisted_perspective_broker_client.py
['test.log']
```

Вывод представляет собой список файлов в указанном каталоге, именно это мы и ожидали получить.

Этот сценарий выглядит несколько сложнее, чем можно было ожидать для такого простого примера RPC. Серверный сценарий выглядит сопоставимым. Создание клиента выгдядит несколько перегруженным из-за объединения функций обратного вызова в конвейер, создания объекта отложенного действия, реакторов и фабрик. Но это был очень простой пример. Преимущества платформы Twisted проявляются особенно ярко, когда задача, которую требуется решить, имеет более высокий уровень сложности.

В примере 5.16 представлена немного модифицированная версия только что продемонстрированного клиента брокера перспективы. Вместо удаленной функции `ls` он вызывает удаленную функцию `ls_boom`. Этот пример демонстрирует, как производится обслуживание исключений на стороне клиента и сервера.

Пример 5.16. Клиент брокера перспективы платформы Twisted – обработка ошибок

```
#!/usr/bin/python

from twisted.spread import pb
from twisted.internet import reactor

def handle_err(reason):
    print "an error occurred", reason
    reactor.stop()

def call_ls(def_call_obj):
    return def_call_obj.callRemote('ls_boom', '/foo')

def print_ls(print_result):
    print print_result
    reactor.stop()

if __name__ == '__main__':
    factory = pb.PBClientFactory()
    reactor.connectTCP("localhost", 9876, factory)
    d = factory.getRootObject()
    d.addCallback(call_ls)
    d.addCallback(print_ls)
    d.addErrback(handle_err)
    reactor.run()
```

Ниже показано, что произошло, когда мы запустили этот сценарий:

```
jmjones@dinkgutsy:code$ python twisted_perspective_broker_client_boom.py an
error occurred [Failure instance: Traceback from remote host -- Traceback
unavailable
]
```

И на стороне сервера:

```
Peer will receive following PB traceback:
Traceback (most recent call last):
...
<большой объем диагностической информации>
...
    state = method(*args, **kw)
  File "twisted_perspective_broker_server.py", line 13, in remote_ls_boom
    return os.listdir(directory)
exceptions.OSError: [Errno 2] No such file or directory: '/foo'
(exceptions.OSError: [Errno 2] Нет такого файла или каталога: '/foo')
```

Конкретное сообщение об ошибке появилось на стороне сервера, а не на стороне клиента. На стороне клиента мы лишь увидели, что произошла какая-то ошибка. Если бы Pyro или XML-RPC вели себя подобным образом, мы посчитали бы, что это недостаток. Но ведь наш обработчик ошибки был вызван в клиентском сценарии, реализованном на платформе Twisted. Так как эта модель программирования (основанная на событиях) отличается от модели программирования, применяемой при использовании Pyro и XML-RPC, мы предполагаем, что обработка ошибок будет производиться иначе, и программный код брокера перспективы сделал именно то, что мы должны были ожидать от него.

Здесь мы представили вашему вниманию даже меньше, чем вершину айсберга Twisted. На первых порах работа с платформой Twisted может оказаться достаточно сложным делом из-за такой широты возможностей этого проекта и подходов к решению задач, так не похожих на то, к чему привыкло большинство из нас. Платформа Twisted определенно заслуживает внимательного изучения и включения ее в свой арсенал.

Scapy

Если вам доставляет удовольствие писать программный код для работы с сетью, вы полюбите Scapy. Scapy – это невероятно удобная интерактивная программа и библиотека манипулирования сетевыми пакетами. Scapy позволяет исследовать сеть, производить сканирование, производить трассировку маршрутов и выполнять зондирование. Более того, для Scapy имеется превосходная документация. Если вам понравилось это вступление, вам следует приобрести книгу, описывающую Scapy более подробно.

Первое, что следует отметить о Scapy, это то, что к моменту написания этих строк данный программный продукт распространялся в виде единственного файла. Вы можете загрузить последнюю версию Scapy по адресу: *http://hg.secdev.org/scapy/raw-file/tip/scapy.py*. После этого вы сможете запускать Scapy как самостоятельную программу или импортировать и использовать этот продукт как библиотеку. Для начала воспользуемся им как интерактивной программой. Пожалуйста, имейте в виду, что программу Scapy придется запускать с привилегия-

ми суперпользователя root, так как ей требуется получить привилегированный доступ к сетевым интерфейсам.

После того как вы загрузите и установите Scapy, вы увидите следующее:

```
Welcome to Scapy (1.2.0.2)
>>>
```

В этой программе вы можете делать все, что обычно делаете в интерактивной оболочке интерпретатора Python, и дополнительно в ваше распоряжение поступают специальные команды Scapy. Первое, что мы сделаем, это вызовем функцию `ls()` в Scapy, которая выводит все доступные уровни:

```
>>> ls()
ARP        : ARP
ASN1_Packet : None
BOOTP      : BOOTP
CookedLinux : cooked linux
DHCP       : DHCP options
DNS        : DNS
DNSQR      : DNS Question Record
DNSRR      : DNS Resource Record
Dot11      : 802.11
Dot11ATIM  : 802.11 ATIM
Dot11AssoReq : 802.11 Association Request
Dot11AssoResp : 802.11 Association Response
Dot11Auth  : 802.11 Authentication
[обрезано]
```

Мы обрезали вывод, потому что он слишком объемный. Ниже мы выполнили рекурсивный запрос DNS имени *www.oreilly.com*, использовав общественный сервер DNS Калифорнийского политехнического университета (Caltech University):

```
>>> sr1(IP(dst="131.215.9.49")/UDP()/DNS(rd=1,qd=DNSQR(qname="
www.oreilly.com")))
Begin emission:
Finished to send 1 packets.
...*
Received 4 packets, got 1 answers, remaining 0 packets
IP version=4L ihl=5L tos=0x0 len=223 id=59364 flags=DF
  frag=0L ttl=239 proto=udp chksum=0xb1e src=131.215.9.49 dst=10.0.1.3
  options=''
|UDP sport=domain dport=domain len=203 chksum=0x843 |
DNS id=0 qr=1L opcode=QUERY aa=0L tc=0L rd=1L ra=1L z=0L
  rcode=ok qdcount=1 ancount=2 nscount=4 arcount=3 qd=
DNSQR qname='www.oreilly.com.' qtype=A qclass=IN |>
  an=DNSRR rrname='www.oreilly.com.' type=A rclass=IN ttl=21600
rdata='208.201.239.36'
[обрезано]
```

Затем выполнили трассировку маршрута:

```
>>> ans,unans=sr(IP(dst="oreilly.com",
>>> ttl=(4,25),id=RandShort())/TCP(flags=0x2))
Begin emission:
..............*Finished to send 22 packets.
*..........*********.***.***.*.*.*.*.*
Received 54 packets, got 22 answers, remaining 0 packets
>>> for snd, rcv in ans:
...     print snd.ttl, rcv.src, isinstance(rcv.payload, TCP)
...
[обрезано]
20 208.201.239.37 True
21 208.201.239.37 True
22 208.201.239.37 True
23 208.201.239.37 True
24 208.201.239.37 True
25 208.201.239.37 True
```

Программе Scapy также под силу воспроизводить содержимое пакетов, на манер утилиты tcpdump:

```
>>> sniff(iface="en0", prn=lambda x: x.show())
###[ Ethernet ]###
dst= ff:ff:ff:ff:ff:ff
src= 00:16:cb:07:e4:58
type= IPv4
###[ IP ]###
version= 4L
ihl= 5L
tos= 0x0
len= 78
id= 27957
flags=
frag= 0L
ttl= 64
proto= udp
chksum= 0xf668
src= 10.0.1.3
dst= 10.0.1.255
options= ''
[обрезано]
```

Кроме того, возможно реализовать трассировку маршрутов в графическом режиме, если в системе установлены graphviz и imagemagic. Следующий пример взят из официальной документации к Scapy:

```
>>> res,unans = traceroute(["www.microsoft.com","www.cisco.com",
  "www.yahoo.com","www.wanadoo.fr","www.pacsec.com"
],dport=[80,443],maxttl=20,
  retry=-2)
Begin emission:
**************************************************************************
```

```
 Finished to send 200 packets.
******************Begin emission:
****************************************Finished to send 110 packets.
************************************************************Begin emission:
Finished to send 5 packets.
Begin emission:
Finished to send 5 packets.

Received 195 packets, got 195 answers, remaining 5 packets
193.252.122.103:tcp443 193.252.122.103:tcp80 198.133.219.25:tcp443
  198.133.219.25:tcp80 207.46.193.254:tcp443 207.46.193.254:tcp80
  69.147.114.210:tcp443 69.147.114.210:tcp80 72.9.236.58:tcp443
  72.9.236.58:tcp80
```

Теперь из полученных результатов можно создать неплохой график:

```
>>> res.graph()
>>> res.graph(type="ps",target="| lp")
>>> res.graph(target="> /tmp/graph.svg")
```

Теперь, если у вас в системе установлены graphviz и imagemagic, вы будете поражены красотой графической визуализации!

Однако истинная прелесть Scapy проявляется при создании своих собственных инструментов командной строки и сценариев. В следующем разделе мы посмотрим на Scapy как на библиотеку.

Создание сценариев с использованием Scapy

Теперь, когда с помощью Scapy мы можем создавать нечто существенное, мы покажем реализацию такого интересного инструмента, как arping. Сначала рассмотрим платформо-зависимую реализацию инструмента arping:

```
#!/usr/bin/env python
import subprocess
import re
import sys

def arping(ipaddress="10.0.1.1"):
    """Функция arping принимает IP-адрес хоста или сети,
       возвращает вложенный список адресов mac/ip"""

    #Предполагается, что arping используется в Red Hat Linux
    p = subprocess.Popen("/usr/sbin/arping -c 2 %s" % ipaddress, shell=True,
                         stdout=subprocess.PIPE)
    out = p.stdout.read()
    result = out.split()
    #pattern = re.compile(":")
    for item in result:
        if ':' in item:
            print item
```

```
if __name__ == '__main__':
    if len(sys.argv) > 1:
        for ip in sys.argv[1:]:
            print "arping", ip
            arping(ip)
    else:
        arping()
```

А теперь посмотрим, как с помощью Scapy можно реализовать то же самое, но платформо-независимым способом:

```
#!/usr/bin/env python
from scapy import srp,Ether,ARP,conf
import sys

def arping(iprange="10.0.1.0/24"):
    """Функция arping принимает IP-адрес хоста или сети,
       возвращает вложенный список адресов mac/ip"""

    conf.verb=0
    ans,unans=srp(Ether(dst="ff:ff:ff:ff:ff:ff")/ARP(pdst=iprange),
              timeout=2)

    collection = []
    for snd, rcv in ans:
        result = rcv.sprintf(r"%ARP.psrc% %Ether.src%").split()
        collection.append(result)
    return collection

if __name__ == '__main__':
    if len(sys.argv) > 1:
        for ip in sys.argv[1:]:
            print "arping", ip
            print arping(ip)
    else:
        print arping()
```

Результатом работы сценария является весьма полезная информация, которая содержит MAC- и IP-адреса всех узлов подсети:

```
# sudo python scapy_arp.py
[['10.0.1.1', '00:00:00:00:00:10'], ['10.0.1.7', '00:00:00:00:00:12'],
['10.0.1.30', '00:00:00:00:00:11'], ['10.0.1.200', '00:00:00:00:00:13']]
```

Эти примеры позволяют получить представление о том, насколько простой и удобной в использовании является Scapy.

6

Данные

Введение

Управление данными, файлами и каталогами – это одна из причин, по которым ИТ-организациям необходимы системные администраторы. У какого системного администратора не возникало необходимости обрабатывать все файлы в дереве каталогов, отыскивать или заменять некоторый текст, и если вам еще не пришлось писать сценарий, который переименовывает все файлы в дереве каталогов, скорее всего это ожидает вас в будущем. Эти умения составляют суть деятельности системного администратора или, по крайней мере, хорошего системного администратора. В этой главе мы сосредоточим свое внимание на данных, файлах и каталогах.

Сисадмины постоянно должны перегонять данные из одного места в другое. Ежедневное перемещение данных у одних системных администраторов составляет большую часть их работы, у других меньшую. В индустрии производства мультипликационных фильмов постоянная «перегонка» данных из одного места в другое является необходимым условием, потому что для производства цифровых фильмов требуются терабайты и терабайты пространства. Различные требования предъявляются к операциям ввода/вывода на дисковые накопители, исходя из качества и разрешения изображения, просматриваемого в каждый конкретный момент времени. Если данные необходимо «перегонять» на жесткий диск для просмотра, чтобы к ним был постоянный доступ в ходе оцифровки, то объектами перемещения будут «свежие» несжатые или с незначительной степенью сжатия файлы изображений с высоким разрешением. Необходимость перемещения файлов обусловлена тем, что в анимационной индустрии вообще используются два типа накопителей. Существуют недорогие, емкие, медленные, надежные накопители и быстрые, дорогостоящие накопители, которые нередко

представляют собой JBOD («just a bunch of disks» – простой дисковый массив), объединенные в массив RAID 0 для обеспечения большей производительности. Системного администратора, которому прежде всего приходится иметь дело с данными, в киноиндустрии часто называют «погонщиком данных».

Погонщик данных должен постоянно перемещать и переносить новые данные из одного места в другое. Часто для этого используются такие утилиты, как rsync, scp или mv. Эти простые, но мощные инструменты могут использоваться в сценариях на языке Python для выполнения самых невероятных действий.

С помощью стандартной библиотеки языка Python можно делать потрясающие вещи без дополнительных затрат. Преимущества стандартной библиотеки состоят в том, что ваши сценарии перемещения данных будут работать везде, независимо от наличия платформозависимой версии, например, утилиты `tar`.

Кроме того, не забывайте про резервное копирование. Существует масса сценариев и приложений резервного копирования, для создания которых требуется смехотворный объем программного кода на языке Python. Мы хотим предупредить вас, что создание дополнительных тестов для проверки программного кода, выполняющего резервное копирование, не только желательно, но и необходимо. Вы обязательно должны провести как модульное, так и функциональное тестирование, если вы используете собственные сценарии резервного копирования.

Кроме того, часто бывает необходимо выполнить обработку данных до, после или в процессе перемещения. Конечно, Python прекрасно подходит для решения и таких задач. Инструмент дедупликации, то есть инструмент, который отыскивает дубликаты файлов и выполняет некоторые действия над ними, очень полезно иметь под рукой, поэтому мы покажем, как создать его. Это один из примеров работы с непрекращающимся потоком данных, с чем часто приходится сталкиваться системным администраторам.

Использование модуля OS для взаимодействия с данными

Если вам когда-нибудь приходилось создавать кросс-платформенные сценарии командной оболочки, вы по достоинству оцените то обстоятельство, что модуль OS предоставляет переносимый прикладной интерфейс доступа к системным службам. В Python 2.5 модуль OS содержит более 200 методов, многие из которых предназначены для работы с данными. В этом разделе мы рассмотрим многие из методов этого модуля, которые пригодятся системным администраторам, которым часто приходится иметь дело с данными.

Всякий раз, когда приходится исследовать новый модуль, оболочка IPython оказывается незаменимым инструментом для этого, поэтому давайте начнем наше путешествие по модулю OS с помощью оболочки IPython, в которой будем выполнять действия, наиболее часто встречающиеся в практике. В примере 6.1 показано, как это делается.

Пример 6.1. Исследование методов модуля OS, наиболее часто используемых при работе с данными

```
In [1]: import os

In [2]: os.getcwd()
Out[2]: '/private/tmp'

In [3]: os.mkdir("/tmp/os_mod_explore")

In [4]: os.listdir("/tmp/os_mod_explore")
Out[4]: []

In [5]: os.mkdir("/tmp/os_mod_explore/test_dir1")

In [6]: os.listdir("/tmp/os_mod_explore")
Out[6]: ['test_dir1']

In [7]: os.stat("/tmp/os_mod_explore")
Out[7]: (16877, 6029306L, 234881026L, 3, 501, 0, 102L,
1207014425, 1207014398, 1207014398)

In [8]: os.rename("/tmp/os_mod_explore/test_dir1",
"/tmp/os_mod_explore/test_dir1_renamed")

In [9]: os.listdir("/tmp/os_mod_explore")
Out[9]: ['test_dir1_renamed']

In [10]: os.rmdir("/tmp/os_mod_explore/test_dir1_renamed")

In [11]: os.rmdir("/tmp/os_mod_explore/")
```

В этом примере, после того как был импортирован модуль OS, в строке [2] мы получили имя текущего рабочего каталога, затем в строке [3] создали новый каталог. После этого в строке [4] с помощью метода `os.listdir()` было получено содержимое этого вновь созданного каталога. Затем мы воспользовались методом `os.stat()`, который похож на команду `stat` в Bash, а затем в строке [8] переименовали каталог. В строке [9] мы убедились, что каталог был переименован, и после этого мы удалили все созданные нами каталоги с помощью метода `os.rmdir()`.

Этот пример ни в коем случае нельзя считать исчерпывающим исследованием модуля OS. Кроме этого существует большое число методов, которые могут вам пригодиться при работе с данными, включая методы изменения прав доступа и методы создания символических ссылок. Чтобы познакомиться с перечнем доступных методов модуля OS, обращайтесь к документации для своей версии Python или воспользуйтесь функцией дополнения по клавише табуляции в оболочке IPython.

Копирование, перемещение, переименование и удаление данных

Во вводном разделе главы мы говорили о перегонке данных, кроме того у вас уже есть некоторое представление о том, как можно использовать модуль OS, поэтому теперь мы можем сразу перейти на более высокий уровень – к модулю `shutil`, который предназначен для работы с более крупномасштабными элементами данных. Модуль `shutil` содержит методы копирования, перемещения, переименования и удаления данных, как и модуль OS, но эти действия могут выполняться над целыми деревьями данных.

Исследование модуля `shutil` в оболочке IPython – это самый увлекательный способ знакомства с ним. В примере ниже мы будем использовать метод `shutil.copytree()`, но в этом модуле имеется множество других методов копирования, принцип действия которых несколько отличается. Чтобы увидеть, в чем заключается разница между различными методами копирования, обращайтесь к документации по стандартной библиотеке языка Python. Взгляните на пример 6.2.

Пример 6.2. Использование модуля shutil для копирования дерева данных

```
In [1]: import os

In [2]: os.chdir("/tmp")
In [3]: os.makedirs("test/test_subdir1/test_subdir2")

In [4]: ls -lR
total 0
drwxr-xr-x 3 ngift wheel 102 Mar 31 22:27 test/

./test:
total 0
drwxr-xr-x 3 ngift wheel 102 Mar 31 22:27 test_subdir1/

./test/test_subdir1:
total 0
drwxr-xr-x 2 ngift wheel 68 Mar 31 22:27 test_subdir2/

./test/test_subdir1/test_subdir2:

In [5]: import shutil

In [6]: shutil.copytree("test", "test-copy")

In [19]: ls -lR
total 0
drwxr-xr-x 3 ngift wheel 102 Mar 31 22:27 test/
drwxr-xr-x 3 ngift wheel 102 Mar 31 22:27 test-copy/

./test:
total 0
drwxr-xr-x 3 ngift wheel 102 Mar 31 22:27 test_subdir1/
```

```
./test/test_subdir1:
total 0
drwxr-xr-x 2 ngift wheel 68 Mar 31 22:27 test_subdir2/

./test/test_subdir1/test_subdir2:

./test-copy:
total 0
drwxr-xr-x 3 ngift wheel 102 Mar 31 22:27 test_subdir1/

./test-copy/test_subdir1:
total 0
drwxr-xr-x 2 ngift wheel 68 Mar 31 22:27 test_subdir2/

./test-copy/test_subdir1/test_subdir2:
```

Очевидно, что это очень простые и, вместе с тем, невероятно полезные действия, а кроме того, вы легко можете использовать подобный программный код внутри более сложного, кросс-платформенного сценария, выполняющего перемещение данных. Первое, что приходит в голову, – подобный программный код можно использовать для перемещения данных из одной файловой системы в другую по определенному событию. При производстве мультипликационных фильмов часто бывает необходимо дождаться завершения работы над последними кадрами, чтобы потом преобразовать их в последовательность, пригодную для редактирования.

Мы могли бы написать сценарий, который в качестве задания для планировщика cron дожидается, пока в каталоге появится «x» кадров. После того как сценарий обнаружит, что в каталоге находится необходимое число кадров, он мог бы переместить этот каталог в другой каталог, где эти кадры будут подвергнуты обработке, или просто переместить их на другой накопитель, достаточно быстрый, чтобы иметь возможность воспроизводить несжатый фильм с высоким разрешением.

Однако модуль `shutil` может не только копировать файлы, в нем также имеются методы для перемещения и удаления деревьев данных. В примере 6.3 демонстрируется возможность перемещения нашего дерева, а в примере 6.4 – возможность его удаления.

Пример 6.3. Перемещение дерева данных с помощью модуля shutil

```
In [20]: shutil.move("test-copy", "test-copy-moved")

In [21]: ls -lR
total 0
drwxr-xr-x 3 ngift wheel 102 Mar 31 22:27 test/
drwxr-xr-x 3 ngift wheel 102 Mar 31 22:27 test-copy-moved/

./test:
total 0
drwxr-xr-x 3 ngift wheel 102 Mar 31 22:27 test_subdir1/

./test/test_subdir1:
```

```
total 0
drwxr-xr-x 2 ngift wheel 68 Mar 31 22:27 test_subdir2/

./test/test_subdir1/test_subdir2:

./test-copy-moved:
total 0
drwxr-xr-x 3 ngift wheel 102 Mar 31 22:27 test_subdir1/

./test-copy-moved/test_subdir1:
total 0
drwxr-xr-x 2 ngift wheel 68 Mar 31 22:27 test_subdir2/

./test-copy-moved/test_subdir1/test_subdir2:
```

Пример 6.4. Удаление дерева данных с помощью модуля shutil

```
In [22]: shutil.rmtree("test-copy-moved")

In [23]: shutil.rmtree("test-copy")
In [24]: ll
```

Перемещение дерева данных является более впечатляющей операцией, чем удаление, поскольку после удаления нам нечего демонстрировать. Многие из этих простых примеров можно было бы объединить с другими действиями в более сложные сценарии. Одна из разновидностей сценариев, которая могла бы быть полезна на практике, – это сценарий резервного копирования, копирующий дерево каталогов на сетевой диск и затем создающий архив, имя которого включает текущие дату и время. К счастью, у нас имеется пример, реализующий на языке Python именно эти действия, который приводится в разделе этой главы, посвященном резервному копированию.

Работа с путями, каталогами и файлами

Невозможно говорить о работе с данными, не принимая во внимание пути, каталоги и файлы. Любой системный администратор должен уметь написать сценарий, который производит обход каталога, выполняет поиск по условию и затем каким-нибудь образом обрабатывает результат. Мы опишем некоторые интересные способы, позволяющие это сделать.

Как всегда, все необходимые для выполнения задания инструменты можно найти в стандартной библиотеке языка Python. Язык Python не зря пользуется репутацией «батарейки входят в комплект поставки». В примере 6.5 демонстрируется, как создать сценарий обхода каталога, содержащий функции, которые явно возвращают файлы, каталоги и пути.

Пример 6.5. Сценарий обхода каталога

```
import os
path = "/tmp"
```

```
def enumeratepaths(path=path):
    """Возвращает пути ко всем файлам в каталоге в виде списка"""
    path_collection = []
    for dirpath, dirnames, filenames in os.walk(path):
        for file in filenames:
            fullpath = os.path.join(dirpath, file)
            path_collection.append(fullpath)

    return path_collection

def enumeratefiles(path=path):
    """Возвращает имена всех файлов в каталоге в виде списка"""
    file_collection = []
    for dirpath, dirnames, filenames in os.walk(path):
        for file in filenames:
            file_collection.append(file)

    return file_collection

def enumeratedir(path=path):
    """Возвращает имена всех подкаталогов в каталоге в виде списка"""
    dir_collection = []
    for dirpath, dirnames, filenames in os.walk(path):
        for dir in dirnames:
            dir_collection.append(dir)

    return dir_collection

if __name__ == "__main__":
    print "\nRecursive listing of all paths in a dir:"
    for path in enumeratepaths():
        print path

    print "\nRecursive listing of all files in dir:"
    for file in enumeratefiles():
        print file

    print "\nRecursive listing of all dirs in dir:"
    for dir in enumeratedir():
        print dir
```

На ноутбуке, работающем под управлением Mac OS, вывод этого сценария выглядит, как показано ниже:

```
[ngift@Macintosh-7][H:12022][J:0]# python enumarate_file_dir_path.py

Recursive listing of all paths in a dir:
/tmp/.aksusb
/tmp/ARD_ABJMMRT
/tmp/com.hp.launchport
/tmp/error.txt
/tmp/liten.py
/tmp/LitenDeplicationReport.csv
/tmp/ngift.liten.log
/tmp/hsperfdata_ngift/58920
```

```
/tmp/launch-h36okI/Render
/tmp/launch-qy1S9C/Listeners
/tmp/launch-RTJzTw/:0
/tmp/launchd-150.wDvODl/sock

Recursive listing of all files in dir:
.aksusb
ARD_ABJMMRT
com.hp.launchport
error.txt
liten.py
LitenDeplicationReport.csv
ngift.liten.log
58920
Render
Listeners
:0
sock

Recursive listing of all dirs in dir:
.X11-unix
hsperfdata_ngift
launch-h36okI
launch-qy1S9C
launch-RTJzTw
launchd-150.wDvODl
ssh-YcE2t6Pfn0
```

Небольшое примечание к предыдущему фрагменту программного кода: метод `os.walk()` возвращает объект-генератор, благодаря которому вы сможете выполнить обход дерева каталогов самостоятельно:

```
In [2]: import os

In [3]: os.walk("/tmp")
Out[3]: [generator object at 0x508e18]
```

Вот как это выглядит при вызове метода в оболочке IPython. Вы можете заметить, что наличие генератора дает нам возможность использовать его метод `path.next()`. Мы не будем углубляться в обсуждение генераторов, но вы должны знать, что метод `os.walk()` возвращает объект-генератор. Генераторы очень полезны для системного программирования. Все, что вам необходимо знать о генераторах, вы найдете на сайте Дэвида Бизли (David Beazely) по адресу: *http://www.dabeaz.com/generators/*.

```
In [2]: import os

In [3]: os.walk("/tmp")
Out[3]: [generator object at 0x508e18]

In [4]: path = os.walk("/tmp")

In [5]: path.
```

```
path.__class__         path.__init__      path.__repr__     path.gi_running
path.__delattr__       path.__iter__      path.__setattr__  path.next
path.__doc__           path.__new__       path.__str__      path.send
path.__getattribute__  path.__reduce__    path.close        path.throw
path.__hash__          path.__reduce_ex__ path.gi_frame
```

```
In [5]: path.next()
Out[5]:
('/tmp',
 ['.X11-unix',
  'hsperfdata_ngift',
  'launch-h36okI',
  'launch-qy1S9C',
  'launch-RTJzTw',
  'launchd-150.wDvODl',
  'ssh-YcE2t6PfnO'],
 ['.aksusb',
  'ARD_ABJMMRT',
  'com.hp.launchport',
  'error.txt',
  'liten.py',
  'LitenDeplicationReport.csv',
  'ngift.liten.log'])
```

Вскоре мы познакомимся с генераторами поближе, но сначала создадим модуль, обладающий прозрачным прикладным интерфейсом, с помощью которого можно будет получать файлы, каталоги и пути.

Теперь, когда мы выполнили основную реализацию задачи обхода каталога, попробуем создать объектно-ориентированный модуль, чтобы впоследствии его легко можно было импортировать и использовать. Модуль с жестко заданными исходными данными получился бы короче, но универсальный модуль, который потом можно будет использовать в разных сценариях, облегчит нам жизнь гораздо существеннее. Взгляните на пример 6.6.

Пример 6.6. Модуль многократного использования для обхода каталога

```
import os

class diskwalk(object):
    """Интерфейс доступа к коллекциям, получаемым при обходе каталога """
    def __init__(self, path):
        self.path = path

    def enumeratePaths(self):
        """Возвращает пути ко всем файлам в каталоге в виде списка"""
        path_collection = []
        for dirpath, dirnames, filenames in os.walk(self.path):
            for file in filenames:
                fullpath = os.path.join(dirpath, file)
                path_collection.append(fullpath)

        return path_collection
```

```
    def enumerateFiles(self):
        """Возвращает имена всех файлов в каталоге в виде списка"""
        file_collection = []
        for dirpath, dirnames, filenames in os.walk(self.path):
            for file in filenames:
                file_collection.append(file)

        return file_collection

    def enumerateDir(self):
        """Возвращает имена всех подкаталогов в каталоге в виде списка"""
        dir_collection = []
        for dirpath, dirnames, filenames in os.walk(self.path):
            for dir in dirnames:
                dir_collection.append(dir)

        return dir_collection
```

Как видите, внесением небольших изменений нам удалось создать отличный интерфейс для модификаций в будущем. Главная прелесть этого нового модуля заключается в том, что его можно импортировать в другие сценарии.

Сравнение данных

Сравнение данных – очень важная операция для системного администратора. Вы часто могли задавать себе вопросы: «Какие файлы в этих двух каталогах различны? Сколько копий одного и того же файла существует у меня в системе?» В этом разделе вы найдете способы, которые позволят вам ответить на эти и другие вопросы.

Когда приходится иметь дело с огромными объемами важных данных, часто бывает необходимо сравнить деревья каталогов и файлов, чтобы узнать, какие изменения были внесены. Это становится еще более важным, когда дело доходит до создания сценариев, перемещающих данные. Судный день будет вам гарантирован, если ваш сценарий перемещения больших объемов данных повредит какие-либо критически важные данные.

В этом разделе мы сначала исследуем несколько легковесных методов сравнения файлов и каталогов, а затем перейдем к вычислению и сравнению контрольных сумм файлов. В стандартной библиотеке языка Python имеется несколько модулей, которые помогут выполнить сравнение; мы рассмотрим `filecmp` и `os.listdir`.

Использование модуля filecmp

Модуль `filecmp` содержит функции для быстрого и эффективного сравнения файлов и каталогов. Модуль `filecmp` вызывает функцию `os.stat()` для двух файлов и возвращает значение `True`, если результаты вызова `os.stat()` одни и те же для обоих файлов, и `False`, если полученные ре-

зультаты отличаются. Обычно функция `os.stat()` вызывается, чтобы определить, не используют ли два файла одни и те же индексные узлы на диске и не имеют ли они одинаковые размеры, но сравнение содержимого файлов при этом не производится.

Чтобы полностью понять, как работает модуль `filecmp`, нам потребуется создать три файла. Для этого перейдем в каталог */tmp*, создадим файл с именем *file0.txt* и запишем в него «0». Затем создадим файл с именем *file1.txt* и запишем в него «1». Наконец, создадим файл с именем *file00.txt* и запишем в него «0». Эти файлы будут использоваться в качестве объектов сравнения в следующем фрагменте:

```
In [1]: import filecmp

In [2]: filecmp.cmp("file0.txt", "file1.txt")
Out[2]: False

In [3]: filecmp.cmp("file0.txt", "file00.txt")
Out[3]: True
```

Как видите, функция `cmp()` вернула значение `True` при сравнении файлов *file0.txt* и *file00.txt*, и `False` при сравнении файлов *file1.txt* и *file0.txt*.

Функция `dircmp()` имеет множество атрибутов, которые сообщают о различиях между двумя деревьями каталогов. Мы не будем рассматривать каждый атрибут, но продемонстрируем несколько примеров выполнения действий, которые могут быть вам полезны. Для этого примера в каталоге */tmp* были созданы два подкаталога, в каждый из которых были скопированы файлы из предыдущего примера. В каталоге *dirB* был создан дополнительный файл с именем *file11.txt*, в который была записана строка «11»:

```
In [1]: import filecmp

In [2]: pwd
Out[2]: '/private/tmp'

In [3]: filecmp.dircmp("dirA", "dirB").diff_files
Out[3]: []

In [4]: filecmp.dircmp("dirA", "dirB").same_files
Out[4]: ['file1.txt', 'file00.txt', 'file0.txt']

In [5]: filecmp.dircmp("dirA", "dirB").report()
diff dirA dirB
Only in dirB : ['file11.txt']
Identical files : ['file0.txt', 'file00.txt', 'file1.txt']
```

Возможно, вас удивило, что атрибут `diff_files` не содержит ничего, хотя мы создали файл *file11.txt* с уникальной информацией в нем. Дело в том, что атрибут `diff_files` выявляет различия только между одноименными файлами.

Затем взгляните на результат вывода атрибута `same_files` и обратите внимание, что он сообщает об идентичных файлах в двух каталогах. Наконец, в последнем примере был сгенерирован отчет. Он наглядно сообщает о различиях между двумя файлами. Это был лишь очень краткий обзор возможностей модуля `filecmp`, поэтому мы рекомендуем обратиться к документации в стандартной библиотеке языка Python, чтобы получить полное представление о имеющихся возможностях, для описания которых мы не располагаем достаточным пространством в книге.

Использование os.listdir

Еще один легковесный способ сравнения двух каталогов основан на использовании метода `os.listdir()`. Метод `os.listdir()` можно представлять себе как аналог команды `ls` – он возвращает список обнаруженных файлов. Язык Python поддерживает множество интересных способов работы со списками, поэтому вы можете использовать метод `os.listdir()` для выявления различий между каталогами, просто преобразуя списки во множества и затем вычитая одно множество из другого. Ниже показано, как это делается в оболочке IPython:

```
In [1]: import os

In [2]: dirA = set(os.listdir("/tmp/dirA"))

In [3]: dirA
Out[3]: set(['file1.txt', 'file00.txt', 'file0.txt'])

In [4]: dirB = set(os.listdir("/tmp/dirB"))

In [5]: dirB
Out[5]: set(['file1.txt', 'file00.txt', 'file11.txt', 'file0.txt'])

In [6]: dirA - dirB
Out[6]: set([])

In [7]: dirB-dirA
Out[7]: set(['file11.txt'])
```

В этом примере можно видеть, что мы преобразовали два списка во множества, а затем выполнили операцию вычитания, чтобы выявить различия. Обратите внимание, что в строке **[7]** было получено имя `file11.txt`, потому что `dirB` является надмножеством для `dirA`, но в строке **[6]** был получен пустой результат, потому что множество `dirA` содержит элементы, которые содержатся в множестве `dirB`. При использовании множеств легко можно создать простое объединение двух структур данных, вычитая полные пути в одном каталоге из путей в другом каталоге, и копируя найденные различия. Объединение данных мы рассмотрим в следующем разделе.

Однако этот подход имеет существенные ограничения. Фактическое имя файла часто может вводить в заблуждение, поскольку ничто не ме-

шает иметь файл с нулевым размером, имя которого совпадает с именем файла, имеющим размер 200 Гбайт. В следующем разделе мы представим несколько лучший способ обнаружения различий между каталогами и объединения их содержимого.

Объединение данных

Как быть, когда необходимо не просто сравнить файлы с данными, но еще и объединить два дерева каталогов в одно? Главная проблема состоит в том, чтобы объединить содержимое одного дерева с другим без создания дубликатов файлов.

Вы могли бы просто вслепую скопировать файлы из одного каталога в другой и затем удалить дубликаты файлов, но гораздо эффективнее было бы вообще не создавать дубликаты. Достаточно простое решение этой проблемы состоит в том, чтобы сравнить два каталога с помощью функции `dircmp()` из модуля `filecmp` и затем скопировать уникальные файлы с помощью приема, основанного на использовании `os.listdir`, описанного выше. Наилучшее решение заключается в использовании контрольных сумм MD5, о чем рассказывается в следующем разделе.

Сравнение контрольных сумм MD5

Вычисление контрольной суммы MD5 файла и сравнение ее с контрольной суммой другого файла напоминает стрельбу из гранатомета по движущейся мишени. Такое мощное оружие вводится в действие, когда требуется полная уверенность в своих действиях, хотя 100-процентную гарантию может дать только побайтовое сравнение файлов. В примере 6.7 показана функция, которая принимает путь к файлу и возвращает его контрольную сумму.

Пример 6.7. Вычисление контрольной суммы MD5 файла

```
import hashlib

def create_checksum(path):
    """
    Читает файл. Вычисляет контрольную сумму файла, строку за строкой.
    Возвращает полную контрольную сумму для всего файла.
    """
    fp = open(path)
    checksum = hashlib.md5()
    while True:
        buffer = fp.read(8192)
        if not buffer: break
        checksum.update(buffer)
    fp.close()
    checksum = checksum.digest()
    return checksum
```

Ниже приводится пример использования этой функции в интерактивной оболочке IPython для сравнения двух файлов:

```
In [2]: from checksum import createChecksum

In [3]: if createChecksum("image1") == createChecksum("image2"):
   ...:     print "True"
   ...:
   ...:
True

In [5]: if createChecksum("image1") == createChecksum("image_unique"):
print "True"
   ...:
   ...:
```

В этом примере контрольные суммы файлов сравниваются вручную, но мы вполне можем использовать программный код, написанный ранее, который возвращает список путей, для рекурсивного сравнивания дерева каталогов и получить список дубликатов. Прелесть удобного API состоит в том, что его теперь можно использовать в оболочке IPython с целью тестирования наших решений в интерактивном режиме. Затем, если решение работает, мы можем создать другой модуль. В примере 6.8 приводится программный код, который отыскивает дубликаты файлов.

Пример 6.8. Вычисление контрольных сумм MD5 в дереве каталогов с целью поиска дубликатов файлов

```
In [1]: from checksum import createChecksum

In [2]: from diskwalk_api import diskwalk

In [3]: d = diskwalk('/tmp/duplicates_directory')

In [4]: files = d.enumeratePaths()

In [5]: len(files)
Out[5]: 12

In [6]: dup = []

In [7]: record = {}

In [8]: for file in files:
   compound_key = (getsize(file),create_checksum(file))
   if compound_key in record:
       dup.append(file)
   else:
   record[compound_key] = file

   ...:
   ...:

In [9]: print dup
['/tmp/duplicates_directory/image2']
```

Фрагмент программного кода, который еще не встречался нам в предыдущих примерах, начинается в строке [7]. Здесь мы создали пустой словарь, и затем сохраняем вычисленные контрольные суммы в виде ключей. Благодаря этому легко можно определить, была ли ранее вычислена та или иная контрольная сумма. Если была, мы помещаем файл в список дубликатов. Теперь давайте выделим часть программного кода, которую позднее мы сможем использовать в разных сценариях. В конце концов, это очень удобно. Как это сделать, показано в примере 6.9.

Пример 6.9. Поиск дубликатов

```
from checksum import create_checksum
from diskwalk_api import diskwalk
from os.path import getsize

def findDupes(path = '/tmp'):
    dup = []
    record = {}
    d = diskwalk(path)
    files = d.enumeratePaths()
    for file in files:
        compound_key = (getsize(file),create_checksum(file))
        if compound_key in record:
            dup.append(file)
        else:
            #print "Creating compound key record:", compound_key
            record[compound_key] = file
    return dup

if __name__ == "__main__":
    dupes = findDupes()
    for dup in dupes:
        print "Duplicate: %s" % dup
```

Запустив этот сценарий, мы получили следующий результат:

```
[ngift@Macintosh-7][H:10157][J:0]# python find_dupes.py
Duplicate: /tmp/duplicates_directory/image2
```

Мы надеемся, вы заметили, что этот пример демонстрирует преимущества повторного использования существующего программного кода. Теперь у нас имеется универсальный модуль, получающий путь к дереву каталогов и возвращающий список дубликатов файлов. Это уже само по себе удобно, но мы можем пойти еще дальше и автоматически удалить дубликаты.

Удаление файлов в языке выполняется очень просто – с помощью метода `os.remove()`. Для этого примера у нас имеется множество файлов размером 10 Мбайт в нашем каталоге */tmp*. Попробуем удалить один из них, воспользовавшись методом `os.remove()`:

```
In [1]: import os

In [2]: os.remove("10
10mbfile.0  10mbfile.1  10mbfile.2  10mbfile.3  10mbfile.4
10mbfile.5  10mbfile.6  10mbfile.7  10mbfile.8

In [2]: os.remove("10mbfile.1")

In [3]: os.remove("10
10mbfile.0  10mbfile.2  10mbfile.3  10mbfile.4  10mbfile.5
10mbfile.6  10mbfile.7  10mbfile.8
```

Обратите внимание, как функция дополнения по клавише табуляции в оболочке IPython позволяет увидеть список соответствующих файлов. Вы должны знать, что метод `os.remove()` удаляет файлы, ничего не сообщая и навсегда, что может не всегда соответствовать нашим желаниям. Учитывая это обстоятельство, мы можем реализовать простой метод, который будет удалять дубликаты, и затем расширить его. Поскольку интерактивная оболочка IPython позволяет легко проверить эту идею, мы напишем проверочную функцию прямо в ней и сразу же проверим ее:

```
In [1]: from find_dupes import findDupes

In [2]: dupes = findDupes("/tmp")

In [3]: def delete(file):
   import os
   ...:     print "deleting %s" % file
   ...:     os.remove(file)
   ...:
   ...:

In [4]: for dupe in dupes:
   ...:     delete(dupe)
   ...:
   ...:

In [5]: for dupe in dupes:
   delete(dupe)
   ...:
   ...:
deleting /tmp/10mbfile.2
deleting /tmp/10mbfile.3
deleting /tmp/10mbfile.4
deleting /tmp/10mbfile.5
deleting /tmp/10mbfile.6
deleting /tmp/10mbfile.7
deleting /tmp/10mbfile.8
```

В этом примере мы несколько усложнили свой метод удаления, добавив в него инструкцию `print`, которая выводит имена удаляемых файлов. Мы уже создали достаточно много программного кода, пригодного для многократного использования, поэтому у нас нет никаких причин

останавливаться на достигнутом. Мы можем создать еще один модуль, который будет выполнять различные операции удаления, получая объект типа `file`. Этот модуль даже не требуется привязывать к поиску дубликатов, его можно использовать для удаления любых файлов. Исходный текст модуля приводится в примере 6.10.

Пример 6.10. Модуль delete

```
#!/usr/bin/env python
import os

class Delete(object):
    """Методы удаления, работающие с объектами типа file"""

    def __init__(self, file):
        self.file = file

    def interactive(self):
        """Интерактивный режим удаления"""

       input = raw_input("Do you really want to delete %s [N]/Y" % self.file)
        if input.upper() == "Y":
            print "DELETING: %s" % self.file
            status = os.remove(self.file)
        else:
            print "Skipping: %s" % self.file
        return

    def dryrun(self):
        """Имитация удаления"""

        print "Dry Run: %s [NOT DELETED]" % self.file
        return

def delete(self):

    """Удаляет файл без дополнительных условий"""

    print "DELETING: %s" % self.file
    try:
        status = os.remove(self.file)
    except Exception, err:
        print err
    return status

if __name__ == "__main__":
    from find_dupes import findDupes
    dupes = findDupes('/tmp')
    for dupe in dupes:
        delete = Delete(dupe)
        #delete.dryrun()
        #delete.delete()
        #delete.interactive()
```

В этом модуле имеется три различных метода удаления. Метод удаления в интерактивном режиме запрашивает у пользователя подтверж-

дение для каждого файла, который предполагается удалить. Это может показаться раздражающим, но этот метод обеспечивает хорошую защиту для тех, кто впоследствии будет сопровождать или изменять этот программный код.

Метод пробного режима всего лишь имитирует удаление. И, наконец, имеется метод, который удаляет файлы безвозвратно. В конце модуля можно увидеть закомментированные варианты использования каждого из трех методов. Ниже приводятся примеры каждого из методов в действии:

- Пробный режим

```
ngift@Macintosh-7][H:10197][J:0]# python delete.py
Dry Run: /tmp/10mbfile.1 [NOT DELETED]
Dry Run: /tmp/10mbfile.2 [NOT DELETED]
Dry Run: /tmp/10mbfile.3 [NOT DELETED]
Dry Run: /tmp/10mbfile.4 [NOT DELETED]
Dry Run: /tmp/10mbfile.5 [NOT DELETED]
Dry Run: /tmp/10mbfile.6 [NOT DELETED]
Dry Run: /tmp/10mbfile.7 [NOT DELETED]
Dry Run: /tmp/10mbfile.8 [NOT DELETED]
```

- Интерактивный режим

```
ngift@Macintosh-7][H:10201][J:0]# python delete.py
Do you really want to delete /tmp/10mbfile.1 [N]/YY
DELETING: /tmp/10mbfile.1
Do you really want to delete /tmp/10mbfile.2 [N]/Y
Skipping: /tmp/10mbfile.2
Do you really want to delete /tmp/10mbfile.3 [N]/Y
```

- Удаление

```
[ngift@Macintosh-7][H:10203][J:0]# python delete.py
DELETING:  /tmp/10mbfile.1
DELETING:  /tmp/10mbfile.2
DELETING:  /tmp/10mbfile.3
DELETING:  /tmp/10mbfile.4
DELETING:  /tmp/10mbfile.5
DELETING:  /tmp/10mbfile.6
DELETING:  /tmp/10mbfile.7
DELETING:  /tmp/10mbfile.8
```

Вы можете согласиться, что приемы инкапсуляции, подобные тем, что были продемонстрированы выше, очень удобны, когда приходится иметь дело с данными, потому что вы можете предотвратить возникновение проблем в будущем, абстрагировавшись от конкретной ситуации и решая универсальную задачу. В данном случае нам необходимо было реализовать удаление дубликатов файлов, поэтому был создан модуль, который универсальным способом отыскивает файлы и удаляет их. Мы могли бы создать еще один инструмент, который получает

объект типа `file` и выполняет сжатие файла. И мы действительно вскоре подойдем к этому примеру.

Поиск файлов и каталогов по шаблону

До сих пор мы рассматривали способы обработки каталогов и файлов и такие действия, как поиск дубликатов, удаление каталогов, перемещение каталогов и так далее. Следующий шаг в освоении дерева каталогов состоит в применении поиска по шаблону либо как самостоятельной операции, либо в комбинации с предыдущими приемами. Как и все прочее в языке Python, реализация поиска по шаблону расширения или имени файла выполняется очень просто. В этом разделе мы продемонстрируем несколько общих проблем, связанных с поиском по шаблону, и применим приемы, использовавшиеся ранее, для создания простых, но мощных инструментов.

Очень часто системным администраторам приходится сталкиваться с необходимостью отыскать и удалить, переместить, переименовать или скопировать файлы определенных типов. Самый простой подход к решению этой задачи в языке Python заключается в использовании модуля `fnmatch` или `glob`. Основное отличие между этими двумя модулями заключается в том, что функция `fnmatch()` при сопоставлении имени файла с шаблоном UNIX возвращает значение `True` или `False`, а функция `glob()` возвращает список путей к файлам, имена которых соответствуют шаблону. Для создания более сложных инструментов поиска по шаблону можно использовать регулярные выражения. Об использовании регулярных выражений более подробно рассказывается в главе 3.

В примере 6.11 показано, как используются функции `fnmatch()` и `glob()`. Здесь мы снова повторно использовали программный код, созданный нами ранее, импортировав класс `diskwalk` из модуля `diskwalk_api`.

Пример 6.11. Использование функций fnmatch( ) и glob( ) в интерактивном режиме для поиска файлов

```
In [1]: from diskwalk_api import diskwalk

In [2]: files = diskwalk("/tmp")

In [3]: from fnmatch import fnmatch

In [4]: for file in files:
   ...:     if fnmatch(file,"*.txt"):
   ...:         print file
   ...:
   ...:
/tmp/file.txt

In [5]: from glob import glob

In [6]: import os
```

```
In [7]: os.chdir("/tmp")

In [8]: glob("*")
Out[8]: ['file.txt', 'image.iso', 'music.mp3']
```

В этом примере, после того как мы воспользовались нашим модулем `diskwalk_api`, у нас появился список полных путей к файлам, находящимся в каталоге */tmp*. После этого мы использовали функцию `fnmatch()`, чтобы определить соответствие каждого файла шаблону `"*.txt"`. Функция `glob()` отличается тем, что она сразу выполняет сопоставление с шаблоном и возвращает список имен файлов. Функция `glob()` является более высокоуровневой по отношению к функции `fnmatch()`, но обе они являются незаменимыми инструментами при решении немного разных задач.

Функцию `fnmatch()` особенно удобно использовать в комбинации с программным кодом, создающим фильтр для поиска данных в дереве каталогов. Часто при работе с каталогами бывает необходимо работать с файлами, имена которых соответствуют определенным шаблонам. Чтобы увидеть этот прием в действии, мы попробуем решить классическую задачу системного администрирования по переименованию всех файлов в дереве каталогов, имена которых соответствуют заданному шаблону. Имейте в виду, что переименовывать файлы так же просто, как удалять, сжимать или обрабатывать их. Для решения подобных задач используется простой алгоритм:

1. Получить путь к файлу в каталоге.
2. Выполнить дополнительную фильтрацию – в эту операцию может быть вовлечено несколько фильтров, таких как имя файла, расширение, размер, уникальность и так далее.
3. Выполнить действие над файлом – скопировать, удалить, сжать, прочитать и так далее. Как это делается, показано в примере 6.12.

Пример 6.12. Переименование файлов с расширением .mp3 в файлы с расширением .txt

```
In [1]: from diskwalk_api import diskwalk

In [2]: from shutil import move

In [3]: from fnmatch import fnmatch

In [4]: files = diskwalk("/tmp")

In [5]: for file in files:
            if fnmatch(file, "*.mp3"):
                #здесь можно сделать все, что угодно: удалить, переместить
                #переименовать ...хм-м, переименовать
                move(file, "%s.txt" % file)


In [6]: ls -l /tmp/
total 0
-rw-r--r--  1 ngift  wheel  0 Apr  1 21:50 file.txt
```

```
-rw-r--r--  1 ngift  wheel  0 Apr  1 21:50 image.iso
-rw-r--r--  1 ngift  wheel  0 Apr  1 21:50 music.mp3.txt
-rw-r--r--  1 ngift  wheel  0 Apr  1 22:45 music1.mp3.txt
-rw-r--r--  1 ngift  wheel  0 Apr  1 22:45 music2.mp3.txt
-rw-r--r--  1 ngift  wheel  0 Apr  1 22:45 music3.mp3.txt
```

При использовании программного кода, разработанного ранее, переименование всех файлов с расширением *.mp3* в каталоге уложилось в четыре строки легко читаемого программного кода на языке Python. Если вы один из немногих системных администраторов, кто не прочитал ни одного эпизода из «BOFH» («Bastard Operator From Hell»), то вам не сразу станет очевидно, что можно было бы дальше сделать с этим фрагментом кода.

Представьте, что у вас имеется технологический файловый сервер, который используется исключительно как высокопроизводительное хранилище файлов с далеко не безграничной емкостью. Вы стали замечать, что диски сервера стали часто переполняться, потому что парочка недобросовестных пользователей принялись размещать на них сотни гигабайтов файлов MP3. Конечно, вы могли бы ввести квотирование дискового пространства для пользователей, но нередко квотирование порождает больше проблем, чем решает. Одно из решений состоит в том, чтобы написать сценарий для запуска его из планировщика cron каждую ночь, который будет отыскивать файлы MP3 и выполнять над ними «случайные» операции. По понедельникам он мог бы давать этим файлам расширение *.txt*, по вторникам – сжимать их в ZIP-архивы, по средам – перемещать в каталог */tmp*, по четвергам – удалять их и отсылать владельцу полный список удаленных файлов MP3 по электронной почте. Мы не можем советовать вам сделать это, если, конечно, вы не являетесь владельцем компании, на которую работаете, но для настоящего «чертова ублюдка оператора» этот пример можно считать воплощением мечты.

Обертка для rsync

Как вы уже, наверное, знаете, `rsync` – это инструмент командной строки, первоначально разрабатывавшийся Эндрю Триджеллом (Andrew Tridgell) и Полом Маккерра (Paul Mackerra). В конце 2007 года стала доступна для тестирования версия 3, включающая еще более широкий перечень параметров, чем оригинальная версия.

За эти годы для нас `rsync` превратился в основной инструмент перемещения данных из пункта А в пункт Б. Объем страницы справочного руководства и количество возможных параметров просто поражают, поэтому мы рекомендуем познакомиться с ними поближе. Без преувеличения утилиту `rsync` можно считать уникальным, наиболее полезным инструментом командной строки, из всех, что когда-либо создавались для системных администраторов.

К этому стоит добавить, что язык Python предоставляет несколько способов управления поведением rsync. Одна из проблем, с которой мы столкнулись, состояла в том, чтобы обеспечить копирование данных в запланированное время. Мы не раз попадали в ситуации, когда было необходимо синхронизировать терабайты данных между двумя файловыми серверами настолько быстро, насколько это возможно, но мы совсем не хотели контролировать этот процесс вручную. Это как раз та ситуация, в которой Python действительно может сыграть значимую роль.

С помощью языка Python можно придать утилите rsync немного искусственного интеллекта и настроить ее под свои нужды. В такой ситуации сценарий на языке Python используется как связующий программный код, который заставляет утилиты UNIX выполнять такие вещи, для которых они никогда не предназначались, и благодаря этому вы получаете возможность создавать очень гибкие и легко настраиваемые инструменты. Вы здесь действительно ограничены только вашим воображением. В примере 6.13. приводится очень простой сценарий, который представляет собой обертку для rsync.

Пример 6.13. Простая обертка для rsync

```
#!/usr/bin/env python
#обертка вокруг rsync для синхронизации содержимого двух каталогов
from subprocess import call
import sys

source = "/tmp/sync_dir_A/" #Обратите внимание на завершающий символ слеша
target = "/tmp/sync_dir_B"
rsync = "rsync"
arguments = "-a"
cmd = "%s %s %s %s" % (rsync, arguments, source, target)

def sync():

    ret = call(cmd, shell=True)
    if ret != 0:
        print "rsync failed"
        sys.exit(1)
sync()
```

Этот пример жестко определяет синхронизацию двух каталогов и выводит сообщение об ошибке, если команда не сработала. Однако мы могли бы реализовать нечто более интересное и решить проблему, с которой часто приходится сталкиваться. Нас часто вызывали, чтобы синхронизировать два очень больших каталога, но мы при этом не собирались следить за синхронизацией данных всю ночь. Но, если вы не контролируете процесс синхронизации, вы можете обнаружить, что процесс был прерван на полпути, при этом данные и целая ночь времени были потрачены впустую, а сам процесс синхронизации придется

опять запускать на следующий день. Используя Python, вы можете создать более агрессивную, высокомотивированную команду rsync.

Что могла бы делать высокомотивированная команда rsync? Она могла бы делать то же самое, что и вы, если бы контролировали процесс синхронизации двух каталогов: она могла бы пытаться продолжать синхронизацию до самого конца и затем посылала бы сообщение по электронной почте с описанием того, что было сделано. В примере 6.14 приводится немного более продвинутый сценарий-обертка для rsync.

Пример 6.14. Команда rsync, которая не завершается, пока не выполнит задание

```
#!/usr/bin/env python
#обертка вокруг rsync для синхронизации содержимого двух каталогов
from subprocess import call
import sys
import time

"""эта мотивированная команда rsync будет пытаться синхронизировать
каталоги, пока не синхронизирует их"""

source = "/tmp/sync_dir_A/" #Note the trailing slash
target = "/tmp/sync_dir_B"
rsync = "rsync"
arguments = "-av"
cmd = "%s %s %s %s" % (rsync, arguments, source, target)

def sync():
    while True:
        ret = call(cmd, shell=True)
        if ret !=0:
            print "resubmitting rsync"
            time.sleep(30)
        else:
            print "rsync was succesful"
            subprocess.call("mail -s 'jobs done' bofh@example.com",
                            shell=True)
            sys.exit(0)

sync()
```

Этот сценарий максимально упрощен и содержит жестко определенные данные, но это – пример полезного инструмента, который можно создать для автоматизации чего-то, что вам обычно приходится контролировать вручную. В этот сценарий можно добавить такие особенности, как возможность устанавливать интервал между попытками и ограничивать количество попыток, проверять объем свободного дискового пространства на машине, с которой устанавливается соединение, и так далее.

Метаданные: данные о данных

Большинство системных администраторов начинают вникать в суть дела, когда принимаются интересоваться не только данными, но и данными о данных. Метаданные, или данные о данных, часто могут играть более важную роль, чем сами данные. Например, в кинопроизводстве и в телевидении одни и те же данные часто хранятся в нескольких каталогах внутри файловой системы или даже в разных файловых системах. Слежение за такими данными часто приводит к созданию своего рода системы управления метаданными.

Метаданные – это данные о том, как организованы и как используются определенные файлы, что может быть очень важно для приложения, для процесса производства мультипликационного фильма или для процедуры восстановления из резервной копии. Python также сможет помочь в этом, поскольку на этом языке легко можно реализовать как чтение, так и запись метаданных.

Рассмотрим использование популярного средства ORM (Object-Relational Mapping – объектно-реляционная проекция) SQLAlchemy для создания метаданных о файловой системе. К счастью, для SQLAlchemy имеется очень качественная документация, а кроме того, этот продукт работает с SQLite. На наш взгляд, это потрясающая комбинация, позволяющая разрабатывать собственные решения по управлению метаданными.

В примерах выше мы выполняли обход файловой системы в режиме реального времени, производили запросы и выполняли действия над обнаруженными файлами. Это невероятно удобно, но поиск по крупным файловым системам, содержащим миллионы файлов, отнимет слишком много времени иногда только для того, чтобы выполнить единственную операцию. В примере 6.15 мы покажем, на что могут быть похожи самые простые метаданные, объединив приемы обхода каталогов со средством ORM.

Пример 6.15. Создание метаданных о файловой системе с помощью SQLAlchemy

```
#!/usr/bin/env python
from sqlalchemy import create_engine
from sqlalchemy import Table, Column, Integer, String, MetaData, ForeignKey
from sqlalchemy.orm import mapper, sessionmaker
import os

#путь
path = " /tmp"

#Часть 1: Создание механизма
engine = create_engine('sqlite:///:memory:', echo=False)

#Часть 2: метаданные
metadata = MetaData()
```

```
filesystem_table = Table('filesystem', metadata,
    Column('id', Integer, primary_key=True),
    Column('path', String(500)),
    Column('file', String(255)),
)
metadata.create_all(engine)

#Часть 3: класс отображения
class Filesystem(object):

    def __init__(self, path, file):
        self.path = path
        self.file = file

    def __repr__(self):
        return "[Filesystem('%s','%s')]" % (self.path, self.file)

#Часть 4: функция отображения
mapper(Filesystem, filesystem_table)

#Часть 5: создание сеанса
Session = sessionmaker(bind=engine, autoflush=True, transactional=True)
session = Session()

#Часть 6: обход файловой системы и заполнение базы данных результатами
for dirpath, dirnames, filenames in os.walk(path):
    for file in filenames:
        fullpath = os.path.join(dirpath, file)
        record = Filesystem(fullpath, file)
        session.save(record)

#Часть 7: подтверждение записи данных в базу
session.commit()

#Часть 8: запрос
for record in session.query(Filesystem):
    print "Database Record Number: %s, Path: %s , File: %s " \
    % (record.id,record.path, record.file)
```

Этот сценарий проще представлять себе как последовательность процедур, выполняемых одна за другой. В первой части создается механизм, который в действительности является лишь несколько необычным способом определения базы данных, которая будет использоваться для хранения метаданных. Во второй части определяется экземпляр класса метаданных и создается таблица в базе данных. В третьей части определяется класс, который будет отображаться в только что созданную таблицу базы данных. В четвертой части вызывается функция отображения, которая производит объектно-реляционную проекцию, а попросту – отображает класс в таблицу. В пятой части создается сеанс связи с базой данных. Обратите внимание, что здесь указано несколько именованных аргументов, включая `autoflush` и `transactional`.

Теперь, когда создание объектно-реляционной проекции закончено, в шестой части мы выполняем уже знакомые нам действия – извлекаем

имена файлов и полные пути при обходе дерева каталогов. Однако здесь имеется пара интересных приемов. Обратите внимание, что для каждого пути и имени файла создается отдельная запись, которая затем сохраняется в базе данных. После этого – в седьмой части – мы подтверждаем транзакцию в нашей базе данных, «расположенной в памяти».

Наконец, в восьмой части выполняется запрос – на языке Python, конечно, возвращающий записи, которые мы поместили в базу данных. Этот пример мог бы стать для вас прекрасной возможностью поэкспериментировать в создании собственных решений использования метаданных с применением SQLAlchemy в вашей компании или у клиентов. Этот пример можно расширить такими дополнительными возможностями, как выполнение реляционных запросов или запись результатов в файл и так далее.

Архивирование, сжатие, отображение и восстановление

Действия с большими объемами данных представляют собой проблему, с которой системные администраторы сталкиваются изо дня в день. Для выполнения своей работы они часто используют `tar`, `dd`, `gzip`, `bzip`, `bzip2`, `hdiutil`, `asr` и другие утилиты.

Хотите верьте, хотите нет, но и в этом случае «батарейки входят в комплект поставки» – стандартная библиотека языка Python имеет встроенную поддержку TAR-файлов, zlib-файлов и gzip-файлов. Если вам требуется сжатие и архивирование, значит, у вас не будет никаких проблем, т. к. язык Python предлагает богатый выбор необходимых инструментов. Давайте поближе посмотрим на дедушку всех архиваторов – tar – и увидим, как стандартная библиотека реализует его поддержку.

Использование модуля tarfile для создания архивов TAR

Создать архив TAR очень просто, даже слишком просто. В примере 6.16 мы создаем очень большой файл. Обратите внимание, что синтаксис создания архива намного более простой, чем даже синтаксис использования самой команды `tar`.

Пример 6.16. Создание большого текстового файла

```
In [1]: f = open("largeFile.txt", "w")

In [2]: statement = "This is a big line that I intend to write over and over
again."

In [3]: x = 0
In [4]: for x in xrange(20000):
   ...:     x += 1
```

```
   ...:     f.write("%s\n" % statement)
   ...:
   ...:
In [4]: ls -l
-rw-r--r-- 1 root root 1236992 Oct 25 23:13 largeFile.txt
```

Теперь, когда у нас имеется большой файл, наполненный мусором, передадим его архиватору TAR, как показано в примере 6.17.

Пример 6.17. Архивирование содержимого файла

```
In [1]: import tarfile

In [2]: tar = tarfile.open("largefile.tar", "w")

In [3]: tar.add("largeFile.txt")

In [4]: tar.close()

In [5]: ll
-rw-r--r-- 1 root root 1236992 Oct 25 23:15 largeFile.txt
-rw-r--r-- 1 root root 1236992 Oct 26 00:39 largefile.tar
```

Как видите, был создан обычный архив TAR, причем намного более простым способом, чем с использованием команды `tar`. Этот пример определенно создает прецедент к использованию оболочки IPython для выполнения повседневной работы по системному администрированию.

Несмотря на удобство создания архивов TAR с помощью Python, тем не менее, практически бесполезно упаковывать в архив один-единственный файл. Используя тот же самый прием обхода каталогов, который мы уже столько раз применяли в этой главе, можно упаковать в архив TAR весь каталог */tmp*, для чего достаточно выполнить обход дерева каталогов и добавить в архив каждый файл, находящийся в каталоге */tmp*, как показано в примере 6.18.

Пример 6.18. Архивирование содержимого дерева каталогов

```
In [27]: import tarfile

In [28]: tar = tarfile.open("temp.tar", "w")

In [29]: import os

In [30]: for root, dir, files in os.walk("/tmp"):
   ....:     for file in filenames:
   ....:
KeyboardInterrupt

In [30]: for root, dir, files in os.walk("/tmp"):
   ....:     for file in files:
   ....:         fullpath = os.path.join(root,file)
   ....:         tar.add(fullpath)
   ....:
   ....:

In [33]: tar.close()
```

В том, чтобы добавить в архив содержимое дерева каталогов при его обходе, нет ничего сложного, и это очень неплохой прием, потому что его можно объединить с другими приемами, рассматривавшимися в этой главе. Представьте, что вы архивируете каталог, заполненный мультимедийными файлами. Было бы неразумно архивировать дубликаты, поэтому у вас вполне может появиться желание перед архивированием заменить дубликаты символическими ссылками. Обладая знаниями, полученными в этой главе, вы легко сможете написать сценарий, который сделает это и сэкономит вам немного дискового пространства.

Поскольку создание простых архивов TAR – занятие довольно скучное, давайте приправим его сжатием bzip2, что заставит ваш процессор скулить и жаловаться на то, как много выпало работы на его долю. Алгоритм сжатия bzip2 иногда может оказаться отличной штукой. Посмотрим, насколько впечатляющим он действительно может быть.

Создадим текстовый файл размером 60 Мбайт и сожмем его до 10 Кбайт, как показано в примере 6.19!

Пример 6.19. Создание архива TAR, сжатого по алгоритму bzip2

```
In [1: tar = tarfile.open("largefilecompressed.tar.bzip2", "w|bz2")

In [2]: tar.add("largeFile.txt")

In [3]: ls -h
foo1.txt fooDir1/ largeFile.txt largefilecompressed.tar.bzip2*
foo2.txt fooDir2/ largefile.tar

ln [4]: tar.close()

In [5]: ls -lh
-rw-r--r-- 1 root root 61M Oct 25 23:15 largeFile.txt
-rw-r--r-- 1 root root 61M Oct 26 00:39 largefile.tar
-rwxr-xr-x 1 root root 10K Oct 26 01:02 largefilecompressed.tar.bzip2*
```

Что самое удивительное, алгоритму bzip2 удалось сжать текстовый файл размером 61 Мбайт в 10 Кбайт, хотя мы и смошенничали, используя одни и те же данные снова и снова. Конечно, этот эффект был получен далеко не бесплатно, потому что в системе на базе двухъядерного процессора AMD на это потребовалось несколько минут.

Теперь попробуем двинуться дальше и создать сжатый архив другими доступными способами, начав с gzip. Синтаксис при этом меняется весьма незначительно, как показано в примере 6.20.

Пример 6.20. Создание архива TAR, сжатого по алгоритму gzip

```
In [10]: tar = tarfile.open("largefile.tar.gzip", "w|gz")

In [11]: tar.add("largeFile.txt")

ln [12]: tar.close()
```

```
In [13]: ls -lh
-rw-r--r-- 1 root root  61M Oct 26 01:20 largeFile.txt
-rw-r--r-- 1 root root  61M Oct 26 00:39 largefile.tar
-rwxr-xr-x 1 root root 160K Oct 26 01:24 largefile.tar.gzip*
```

Архив gzip тоже отличается невероятно маленьким размером, уместившись в 160 Кбайт, причем на моей машине сжатый архив TAR был создан за несколько секунд. В большинстве ситуаций это неплохой компромисс.

Использование модуля tarfile для проверки содержимого файлов TAR

Теперь, когда у нас имеется инструмент создания файлов TAR, есть смысл попробовать проверить содержимое файлов TAR. Создать файл TAR – это лишь полдела. Если вы проработали системным администратором достаточно продолжительное время, вам, вероятно, случалось «погореть» с некачественной резервной копией или случалось быть обвиненным в создании некачественной резервной копии.

Чтобы воспроизвести эту ситуацию и подчеркнуть важность проверки архивов TAR, мы поделимся историей о нашем вымышленном друге, которую назовем «Проблема пропавшего архива TAR». Имена, названия и факты являются вымышленными. Любые совпадения с действительностью являются случайными.

Наш друг работал в крупной телестудии в качестве системного администратора и отвечал за поддержку отдела, во главе которого стоял по-настоящему невыдержанный человек. У этого руководителя была репутация неправдивого, импульсивного и невыдержанного человека. Если возникала ситуация, когда этот сумасшедший совершал промах, например не укладывался в оговоренные с клиентом сроки или выполнял свою часть программы не в соответствии с требуемыми характеристиками, он с большим удовольствием лгал и перекладывал ответственность на кого-нибудь другого. Зачастую этим кем-нибудь другим оказывался наш друг, системный администратор.

К сожалению, наш друг отвечал за содержание резервных копий этого сумасшедшего. Ему уже стало казаться, что настало время подыскивать другую работу, но он работал в этой студии уже много лет, у него было много друзей, и он не хотел потерять все из-за этих временных неурядиц. Ему требовалась система, позволяющая убедиться, что он охватил резервированием все данные, и поэтому он ввел регистрационную систему, которая классифицировала содержимое всех архивов TAR, которые автоматически создавались для этого сумасшедшего, так как понимал, что может «погореть», и это лишь вопрос времени, когда сумасшедший опять не уложится в сроки и ему потребуется причина для оправдания.

Однажды нашему другу Вильяму позвонил начальник и сказал: «Вильям, зайдите ко мне немедленно, у нас неприятности с резервными копиями». Вильям сразу же пошел к начальнику и узнал, что этот сумасшедший, Алекс, обвинил Вильяма в повреждении архива со съемкой телешоу, из-за чего произошла задержка с передачей программы клиенту. Срыв Алексом сроков сдачи совершенно вывел Боба, начальника Алекса, из себя.

Начальник сказал Вильяму, что, по словам Алекса, резервная копия содержала только поврежденные файлы и что из-за этого были сорваны сроки подготовки шоу. В ответ Вильям сказал боссу, что был уверен в том, что рано или поздно его обвинят в порче архива и поэтому втайне написал на языке Python сценарий, который проверяет содержимое всех создаваемых им архивов TAR и записывает расширенные сведения об атрибутах файлов до и после резервного копирования. Оказалось, что Алекс так и не приступал к работе над шоу и что в течение нескольких месяцев архивировалась пустая папка.

Когда Алекс был поставлен перед фактами, он быстро пошел на попятную и попытался перевести внимание на другую проблему. К несчастью для Алекса, этот случай стал последней каплей и пару месяцев спустя он исчез с работы. Возможно, он уехал или был уволен, но это уже не важно, наш друг успешно решил проблему пропавшего архива TAR.

Мораль этой истории заключается в том, что, когда приходится иметь дело с резервными копиями, с ними следует обращаться как с ядерным оружием, так как резервные копии могут хранить в себе такие опасности, о которых вы даже не подозреваете.

Ниже демонстрируется несколько способов проверки содержимого файла TAR, созданного ранее:

```
In [1]: import tarfile

In [2]: tar = tarfile.open("temp.tar","r")

In [3]: tar.list()
-rw-r--r-- ngift/wheel 2 2008-04-04 15:17:14 tmp/file00.txt
-rw-r--r-- ngift/wheel 2 2008-04-04 15:15:39 tmp/file1.txt
-rw-r--r-- ngift/wheel 0 2008-04-04 20:50:57 tmp/temp.tar
-rw-r--r-- ngift/wheel 2 2008-04-04 16:19:07 tmp/dirA/file0.txt
-rw-r--r-- ngift/wheel 2 2008-04-04 16:19:07 tmp/dirA/file00.txt
-rw-r--r-- ngift/wheel 2 2008-04-04 16:19:07 tmp/dirA/file1.txt
-rw-r--r-- ngift/wheel 2 2008-04-04 16:19:52 tmp/dirB/file0.txt
-rw-r--r-- ngift/wheel 2 2008-04-04 16:19:52 tmp/dirB/file00.txt
-rw-r--r-- ngift/wheel 2 2008-04-04 16:19:52 tmp/dirB/file1.txt
-rw-r--r-- ngift/wheel 3 2008-04-04 16:21:50 tmp/dirB/file11.txt

In [4]: tar.name
Out[4]: '/private/tmp/temp.tar'

In [5]: tar.getnames()
```

```
Out[5]:
['tmp/file00.txt',
 'tmp/file1.txt',
 'tmp/temp.tar',
 'tmp/dirA/file0.txt',
 'tmp/dirA/file00.txt',
 'tmp/dirA/file1.txt',
 'tmp/dirB/file0.txt',
 'tmp/dirB/file00.txt',
 'tmp/dirB/file1.txt',
 'tmp/dirB/file11.txt']

In [10]: tar.members
Out[10]:
[<TarInfo 'tmp/file00.txt' at 0x109eff0>,
 <TarInfo 'tmp/file1.txt' at 0x109ef30>,
 <TarInfo 'tmp/temp.tar' at 0x10a4310>,
 <TarInfo 'tmp/dirA/file0.txt' at 0x10a4350>,
 <TarInfo 'tmp/dirA/file00.txt' at 0x10a43b0>,
 <TarInfo 'tmp/dirA/file1.txt' at 0x10a4410>,
 <TarInfo 'tmp/dirB/file0.txt' at 0x10a4470>,
 <TarInfo 'tmp/dirB/file00.txt' at 0x10a44d0>,
 <TarInfo 'tmp/dirB/file1.txt' at 0x10a4530>,
 <TarInfo 'tmp/dirB/file11.txt' at 0x10a4590>]
```

Эти примеры показывают, как получить имена файлов, хранящиеся в архиве TAR, чтобы впоследствии иметь возможность их проанализировать в сценарии, проверяющем данные. Извлечение файлов из архивов выполняется ничуть не сложнее. Если вам потребуется извлечь все файлы из архива TAR в текущий рабочий каталог, можно воспользоваться следующей функцией:

```
In [60]: tar.extractall()

drwxrwxrwx 7 ngift wheel 238 Apr 4 22:59 tmp/
```

Если вы чрезвычайно подозрительны, каковыми и должны быть, то вы могли бы реализовать подсчет контрольных сумм MD5 случайных файлов при извлечении их из архива и сравнивать их с соответствующими контрольными суммами, которые были сохранены до упаковки файлов в архив. Это очень эффективный способ убедиться в том, что целостность данных не нарушена.

Ни одно разумное решение не должно основываться на предположении, что архив был создан без ошибок. По крайней мере, хотя бы выборочная проверка архивов должна выполняться автоматически. Но лучше, если сразу после создания каждый архив будет открываться и проверяться.

7

SNMP

Введение

Протокол SNMP может изменить вашу жизнь системного администратора. Отдача от использования SNMP ощущается не так скоро, как от нескольких строк программного кода на языке Python, выполняющих анализ файла журнала, например, но когда инфраструктура SNMP будет настроена, работа с ней начинает удивлять.

В этой главе мы рассмотрим следующие аспекты SNMP: автообнаружение, опрос/мониторинг, создание агентов, управление устройствами и, наконец, интеграцию оборудования средствами SNMP. Безусловно, все это можно реализовать на языке Python.

Если вы не знакомы с SNMP или вам требуется освежить свои знания о SNMP, мы настоятельно рекомендуем прочитать книгу «Essential SNMP» Дугласа Мауро (Douglas Mauro) и Кевина Шмидта (Kevin Schmidt) (O'Reilly) или хотя бы держать ее под рукой. Хороший справочник является основой к истинному пониманию возможностей SNMP. В следующем разделе мы рассмотрим основы SNMP, но глубокое изучение этого протокола выходит далеко за рамки этой книги. В действительности тема использования Python в комплексе с SNMP настолько обширна, что заслуживает отдельной книги.

Краткое введение в SNMP

Обзор SNMP

С высоты 3000 метров SNMP – это протокол управления устройствами в IP-сетях. Как правило, этот протокол работает с портами UDP 161 и 162, хотя вполне возможно использовать и порты TCP. Практически все современные устройства в центрах обработки данных поддержива-

ют работу с протоколом SNMP, а это означает, что имеется возможность управлять не только коммутаторами и маршрутизаторами, но также серверами, принтерами, блоками бесперебойного питания, накопителями и другими устройствами.

Работа протокола SNMP основана на передаче хостам пакетов UDP и ожидании ответов. Таким образом на самом простом уровне производится мониторинг устройств. Тем не менее, протокол SNMP обладает гораздо более широкими возможностями благодаря управляющим устройствам и возможности создания агентов, отвечающих на запросы.

Наиболее типичными примерами того, что возможно с применением SNMP, является мониторинг нагрузки на процессор, использования диска и объема свободной памяти. Этот протокол может также использоваться для управления сетевыми коммутаторами, с его помощью вполне возможно даже выполнять загрузку новых параметров настройки коммутатора. Мало кому известно, что точно так же можно осуществлять мониторинг программного обеспечения, такого как веб-приложения и базы данных. Наконец, имеется поддержка RMON MIB (Remote Monitoring Management Information Base – база управляющей информации для удаленного мониторинга), которая обеспечивает мониторинг «динамики», тогда как в обычном режиме SNMP применяется для мониторинга статических показателей.

Мы уже упомянули аббревиатуру MIB, поэтому сейчас объясним, что это такое. SNMP – это всего лишь протокол, и он не делает никаких предположений о данных. На подконтрольных устройствах выполняется агент, snmpd, у которого имеется перечень объектов, подвергаемых мониторингу. Фактически перечень представляет собой базу управляющей информации, или MIB (Management Information Base). У каждого агента имеется, по крайней мере, одна база MIB, структура которой соответствует спецификациям MIB-II, определяемым в RFC 1213. Базу MIB можно представить себе как файл, который используется для трансляции имен в числа (чем-то похоже на DNS), хотя на самом деле все немного сложнее.

В этом файле находятся описания объектов управления. У каждого объекта имеется три атрибута: имя, тип и синтаксис и данные для передачи. Из них вам чаще всего придется работать с именами. Имена часто еще называют идентификаторами объектов, или OID (Object Identifier). Передавая этот OID агенту, вы тем самым сообщаете, что именно хотели бы получить. Имена имеют две формы представления: числовую и «удобочитаемую». Чаще используется удобочитаемая форма имен, потому что числовые имена имеют большую длину и их сложно запоминать. Одним из самых часто используемых OID является sysDescr. Если вы воспользуетесь инструментом командной строки `snmpwalk`, чтобы получить значение идентификатора sysDescr, вы можете использовать как удобочитаемую, так и числовую форму представления:

```
[root@rhel][H:4461]# snmpwalk -v 2c -c public localhost .1.3.6.1.2.1.1.1.0
SNMPv2-MIB::sysDescr.0 = STRING: Linux localhost
  2.6.18-8.1.15.el5 #1 SMP Mon Oct 22 08:32:04 EDT 2007 i686

[root@rhel][H:4461]# snmpwalk -v 2c -c public localhost sysDescr
SNMPv2-MIB::sysDescr.0 = STRING: Linux localhost
  2.6.18-8.1.15.el5 #1 SMP Mon Oct 22 08:32:04 EDT 2007 i686
```

К этому моменту мы нагрузили вас уймой аббревиатур и RFC, но призываем вас пересилить в себе желание встать и пойти спать. Мы обещаем, что очень скоро исправимся и приступим к разработке программного кода.

Установка и настройка SNMP

Для упрощения дальнейшего повествования мы будем использовать только пакет Net-SNMP и соответствующее расширение Python к нему. Но это не говорит о его большей ценности в сравнении с другими библиотеками SNMP для Python, например PySNMP, используемой в таких продуктах, как TwistwdSNMP и Zenoss. В Zenoss и в Twisted-SNMP библиотека PySNMP используется в асинхронном режиме. Это очень правильный подход, который заслуживает рассмотрения, но у нас просто нет места, чтобы описать оба эти продукта в данной главе.

Говоря в терминах Net-SNMP, мы будем иметь дело с двумя различными прикладными интерфейсами (API). Первый метод состоит в использовании модуля `subprocess`, чтобы «обернуть» инструменты командной строки из пакета Net-SNMP, а второй – в использовании новых расширений для Python. Каждый из этих методов имеет свои преимущества и недостатки в зависимости от среды, в которой они применяются.

В заключение мы также познакомимся с продуктом Zenoss, который представляет собой весьма внушительное решение мониторинга сетей посредством протокола SNMP, полностью реализованное на языке Python и распространяемое с открытыми исходными текстами. При использовании Zenoss нам не придется писать средства управления SNMP с чистого листа и вместо этого мы сможем взаимодействовать с ним посредством его общедоступного API. Кроме того, проект Zenoss предоставляет нам возможность создавать собственные модули для этого продукта, вносить исправления и, наконец, расширять его функциональность.

Чтобы добиться чего-то полезного от SNMP, и в частности от Net-SNMP, его сначала нужно установить. К счастью, большинство операционных систем UNIX и Linux устанавливаются вместе с пакетом Net-SNMP, поэтому, если вам необходимо реализовать мониторинг устройства, для этого достаточно будет выполнить необходимые настройки в конфигурационном файле *snmpd.conf* и запустить демон. Если вы предполагаете разрабатывать на языке Python приложения, использующие пакет

Net-SNMP, о котором идет речь в этой главе, вам необходимо скомпилировать и установить расширения для Python. Если же вы предполагаете просто обертывать команды Net-SNMP, такие как `snmpget`, `snmpwalk`, `snmpdf` и другие, тогда вам ничего не потребуется делать, если сам пакет Net-SNMP уже установлен.

Как вариант, вы можете загрузить виртуальную машину с исходными текстами примеров для этой книги с сайта издательства *http://www.oreilly.com/9780596515829*. Вы можете также обращаться на сайт поддержки книги *www.py4sa.com*, где найдете последнюю информацию о том, как можно опробовать примеры из этого раздела.

Кроме того, мы настроили эту виртуальную машину и с поддержкой Net-SNMP, и с необходимыми расширениями для Python. Вы можете просто использовать эту виртуальную машину для запуска всех примеров. Если мощность вашего компьютера позволяет, вы можете создать несколько копий виртуальной машины и запускать под их управлением другие примеры из этой главы, чтобы имитировать взаимодействия с несколькими компьютерами одновременно.

Если вы решите самостоятельно установить расширения для Python, вам потребуется загрузить с сайта *sourceforge.net* Net-SNMP версии 5.4.x или выше. Расширения в этом пакете не скомпилированы по умолчанию, поэтому вам придется самостоятельно собрать их, следуя инструкциям в каталоге *Python/README*. В двух словах заметим, что вам сначала надо будет скомпилировать эту версию Net-SNMP, а затем запустить сценарий `setyp.py` в каталоге *Python*. Мы считаем, что процедура установки наименее утомительна в дистрибутиве Red Hat Linux, где имеется пакет RPM с исходными текстами. Если вы решили выполнить компиляцию, возможно, вам следует сначала попробовать сделать это в Red Hat, чтобы ознакомиться с самим процессом, а затем приступать к установке в AIX, Solaris, OS X, HP-UX и в других операционных системах. Наконец, если столкнетесь с неприятностями, то для запуска примеров просто воспользуйтесь виртуальной машиной, а порядок компиляции и установки выясните позже.

И еще одно последнее замечание: обязательно выполните команду `setup.py build` и затем `setup.py test`. Это сразу же позволит вам проверить возможность работы с Net-SNMP из Python. В качестве совета: если вы столкнетесь с неприятностями во время компиляции, запустите команду `ldconfig`, как показано ниже:

```
ldconfig -v /usr/local/lib/
```

Если вам случится устанавливать пакет Net-SNMP на стороне клиента, который предполагается подвергнуть мониторингу, вам следует скомпилировать Net-SNMP с параметром Host Resources MIB. Для этого обычно достаточно выполнить следующую команду конфигурирования процесса сборки:

```
./configure -with-mib-modules=host
```

Обратите внимание, что при запуске сценария `configure` он попытается запустить сценарий автоматической настройки. Но вам не обязательно делать это. Часто бывает проще вручную создать свой конфигурационный файл. В Red Hat настройки обычно сохраняются в файле */etc/snmp/snmpd.conf* и имеют примерно следующий вид:

```
syslocation "O'Reilly"
syscontact bofh@oreilly.com
rocommunity public
```

Этого простого файла будет вполне достаточно для опробования примеров в оставшейся части главы и запросов не для третьей версии SNMP. Версия SNMPv3 имеет несколько более сложные настройки и не совсем вписывается в тему данной главы, хотя при этом мы хотели бы заметить, что в производстве лучше использовать SNMPv3, так как версии 2 и 1 не имеют никакой защиты. Это значит, что никогда не следует использовать SNMPv2 и SNMPv1 для передачи запросов через Интернет, поскольку этот трафик может быть перехвачен. Известны случаи высококлассных взломов, которые стали возможны благодаря использованию этих версий.

IPython и Net–SNMP

Если прежде вы никогда не занимались разработкой для SNMP, у вас может возникнуть ощущение, что это не самая приятная работа. Честно говоря, это так и есть. Работа с SNMP чем-то сродни головной боли – из-за высокой сложности протокола, из-за необходимости читать большое число RFC и из-за высоких шансов допустить ошибку. Один из способов попытаться ослабить эту боль состоит в том, чтобы для исследования SNMP и получения навыков обращения с API использовать оболочку IPython.

В примере 7.1 представлен очень короткий фрагмент программного кода для запуска на локальной машине.

Пример 7.1. Использование IPython и Net-SNMP с расширениями Python

```
In [1]: import netsnmp

In [2]: oid = netsnmp.Varbind('sysDescr')

In [3]: result = netsnmp.snmpwalk(oid,
   ...:                           Version = 2,
   ...:                           DestHost="localhost",
   ...:                           Community="public")

Out[4]: ('Linux localhost 2.6.18-8.1.14.el5 #1 SMP Thu Aug 27 12:51:54 EDT
2008 i686',)
```

При исследовании библиотеки очень помогает использование функции дополнения по нажатию клавиши табуляции. В этом примере мы вовсю использовали функцию дополнения в IPython и с ее помощью

создали очень простой запрос SNMPv2. В качестве общего примечания: идентификатор sysDescr, о котором мы уже упоминали ранее, представляет собой очень важный запрос, позволяющий получить базовые характеристики машины. В выводе этого примера можно увидеть нечто похожее, хотя и не идентичное, тому, что выводит команда `uname -a`.

Как будет показано ниже в этой главе, анализ ответа на запрос sysDescr является важной частью исследования центров обработки данных. К сожалению, как и многие составляющие SNMP, этот ответ не совсем точен. Некоторые устройства могут не возвращать никакого ответа, некоторые могут возвращать хоть и полезную, но неполную информацию, например: «Fibre Switch» (оптоволоконный коммутатор), некоторые могут возвращать полную строку идентификации производителя. У нас недостаточно места, чтобы углубляться в детали решения этой проблемы, но заметим, что умение анализировать все эти различия как раз и есть то, на чем большие мальчики зарабатывают деньги.

Как вы уже знаете из главы 2 «IPython», существует возможность создать определение класса или функции в виде отдельного файла прямо из оболочки IPython, переключившись в редактор Vim, выполнив следующую команду:

```
ed some_filename.py
```

После выхода из редактора вы получаете атрибуты созданного модуля в своем пространстве имен и можете вывести их командой `who`. Этот прием очень удобно использовать при работе с SNMP, так как итеративный стиль программирования естественным образом вписывается в эту прикладную область. Давайте двинемся дальше и запишем следующий ниже фрагмент программного кода в файл с именем *snmp.py*, выполнив команду:

```
ed snmp.py
```

В примере 7.2 приводится простой модуль, представляющий собой шаблон создания сеанса с помощью Net-SNMP.

Пример 7.2. Простой модуль создания сеанса с помощью Net-SNMP

```
#!/usr/bin/env python
import netsnmp

class Snmp(object):
    """Простой сеанс SNMP"""
    def __init__(self,
                 oid = "sysDescr",
                 Version = 2,
                 DestHost = "localhost",
                 Community = "public"):
        self.oid = oid
        self.version = Version
```

```
        self.destHost = DestHost
        self.community = Community

    def query(self):
        """Создает запрос SNMP"""
        try:
            result = netsnmp.snmpwalk(self.oid,
                                      Version = self.version,
                                      DestHost = self.destHost,
                                      Community = self.community)
        except Exception, err:
            print err
            result = None
        return result
```

После того как вы сохраните этот файл и введете команду `who`, вы получите следующее:

```
In [2]: who
Snmp netsnmp
```

Теперь, когда у нас имеется объектно-ориентированный интерфейс к SNMP, можно воспользоваться им, чтобы выполнить запрос к локальной машине:

```
In [3]: s = snmp()

In [4]: s.query()
Out[4]: ('Linux localhost 2.6.18-8.1.14.el5 #1 SMP Thu Sep 27 18:58:54 EDT
2007 i686',)

In [5]: result = s.query()

In [6]: len(result)
Out[6]: 1
```

Глядя на этот пример, можно сказать, что с помощью этого модуля легко можно получать результаты, хотя в данном случае мы просто запустили сценарий, в котором жестко определили исходные данные; поэтому теперь попробуем изменить значение объекта OID, чтобы выполнить обход всего поддерева системы:

```
In [7]: s.oid
Out[7]: 'sysDescr'

In [8]: s.oid = ".1.3.6.1.2.1.1"

In [9]: result = s.query()

In [10]: print result
('Linux localhost 2.6.18-8.1.14.el5 #1 SMP Thu Sep 27 18:58:54 EDT 2007 i686',
'.1.3.6.1.4.1.8072.3.2.10', '121219', 'me@localhost.com', 'localhost',
'"My Local Machine"', '0', '.1.3.6.1.6.3.10.3.1.1', '.1.3.6.1.6.3.11.3.1.1',
'.1.3.6.1.6.3.15.2.1.1', '.1.3.6.1.6.3.1',
'.1.3.6.1.2.1.49', '.1.3.6.1.2.1.4', '.1.3.6.1.2.1.50',
'.1.3.6.1.6.3.16.2.2.1', 'The SNMP Management Architecture MIB.',
```

```
'The MIB for Message Processing and Dispatching.', 'The management information
  definitions for the SNMP User-based Security Model.',
'The MIB module for SNMPv2 entities', 'The MIB module for managing TCP
  implementations', 'The MIB module for managing IP and ICMP implementations',
'The MIB module for managing UDP [snip]',
'View-based Access Control Model for SNMP.', '0', '0', '0', '0', '0', '0',
'0', '0')
```

Такой стиль интерактивного и исследовательского программирования делает работу с SNMP более приятной. К этому моменту, если вы чувствуете в себе уверенность, можете приступить к выполнению запросов с другими значениями OID или даже выполнить обход всего дерева MIB. Однако обход полного дерева MIB может занять некоторое время, потому что потребуется выполнить запросы для множества значений OID, поэтому на практике такой подход обычно не используется, так как при этом будут потребляться ресурсы клиентской машины.

Не забывайте, что MIB-II – это всего лишь файл со значениями OID, который входит в состав большинства систем, обладающих поддержкой SNMP. Другие MIB, уникальные для каждого производителя, располагаются в отдельных файлах, к которым может обращаться агент, чтобы вернуть ответ на запрос. Если вы захотите перейти на следующую ступень мастерства, вам придется найти специализированную документацию от производителя с описанием того, в какой базе MIB следует запрашивать тот или иной OID.

Теперь перейдем к использованию особенности оболочки IPython, которая позволяет запускать выполнение заданий в фоновом режиме:

```
In [11]: bg s.query()
Starting job # 0 in a separate thread.

In [12]: jobs[0].status
Out[12]: 'Completed'

In [16]: jobs[0].result
Out[16]:
('Linux localhost 2.6.18-8.1.14.el5 #1 SMP Thu Sep 27 18:58:54 EDT 2007 i686',
'.1.3.6.1.4.1.8072.3.2.10', '121219', 'me@localhost.com', 'localhost',
  '"My Local Machine"',
'0', '.1.3.6.1.6.3.10.3.1.1', '.1.3.6.1.6.3.11.3.1.1',
  '.1.3.6.1.6.3.15.2.1.1', '.1.3.6.1.6.3.1',
'.1.3.6.1.2.1.49', '.1.3.6.1.2.1.4', '.1.3.6.1.2.1.50',
'.1.3.6.1.6.3.16.2.2.1',
'The SNMP Management Architecture MIB.', 'The MIB for Message Processing and
  Dispatching.',
'The management information definitions for the SNMP User-based Security
  Model.',
'The MIB module for SNMPv2 entities', 'The MIB module for managing TCP
implementations',
'The MIB module for managing IP and ICMP implementations', 'The MIB module for
```

```
    managing UDP implementations',
'View-based Access Control Model for SNMP.', '0', '0', '0', '0', '0', '0',
    '0', '0')
```

Прежде чем вы придете в восхищение, разрешите сообщить, что хотя выполнение действий в фоновом режиме прекрасно реализовано в IPython, тем не менее, этот режим может использоваться только при работе с библиотеками, поддерживающими асинхронную модель выполнения. А расширения Python для Net-SNMP работают в синхронном режиме. В двух словах отметим, что вы не сможете писать многопоточный программный код, ожидающий ответа, так как в основе лежат блоки кода на языке C.

К счастью, как будет показано в главе, рассказывающей о процессах и многозадачности, с помощью модуля обработки легко можно создавать дочерние процессы для параллельного выполнения запросов SNMP. В следующем разделе мы рассмотрим проблему создания сценария, который будет автоматически исследовать центр обработки данных.

Исследование центра обработки данных

Одна из наиболее полезных сторон SNMP заключается в использовании этого протокола для исследования центра обработки данных. Проще говоря, в ходе исследования составляется опись устройств, подключенных к сети, и производится сбор информации об этих устройствах. Более детальные виды исследований могут использоваться для выявления связей между собранными данными, например, выяснение точного MAC-адреса, под которым сервер известен коммутатору Cisco, или схемы распределения памяти для оптоволоконного коммутатора Brocade.

В этом разделе мы создадим простой сценарий, который будет отбирать корректные IP-адреса, MAC-адреса, основную информацию, поставляемую протоколом SNMP, и помещать ее в записи. Этот сценарий может использоваться в вашей организации как основа для реализации приложений, выполняющих исследование центра обработки данных. При создании сценария мы будем использовать сведения, которые рассматривались в других главах.

Существует несколько различных алгоритмов исследования, с которыми нам приходилось сталкиваться, но только один из них мы представим вашему вниманию. Суть алгоритма состоит в следующем: послать серию запросов по протоколу ICMP; каждому ответившему устройству послать простой запрос SNMP; проанализировать ответ; продолжить исследование на основе полученных данных. Другой алгоритм подразумевает посылку серии запросов SNMP и сбор ответов с помощью другого процесса, но, как уже говорилось выше, мы сосредоточимся на реализации первого алгоритма. Взгляните на пример 7.3.

Небольшое замечание к программному коду ниже: поскольку библиотека Net-SNMP предусматривает возможность работы только в синхронном режиме, мы создаем дочерние процессы вызовом `subprocess.call()`. Это приводит к возможности появления блокировок. В части использования утилиты `ping` мы могли бы просто использовать `subprocess.Popen`, но чтобы сохранить единообразие, мы используем один и тот же прием как для выполнения запросов SNMP, так и при использовании утилиты `ping`.

Пример 7.3. Простой сценарий исследования центра обработки данных

```
#!/usr/bin/env python
from processing import Process, Queue, Pool
import time
import subprocess
import sys
from snmp import Snmp

q = Queue()
oq = Queue()
#ips = IP("10.0.1.0/24")
ips = ["192.19.101.250", "192.19.101.251", "192.19.101.252",
       "192.19.101.253", "192.168.1.1"]
num_workers = 10

class HostRecord(object):
    """Записи с информацией о хостах"""
    def __init__(self, ip=None, mac=None, snmp_response=None):
        self.ip = ip
        self.mac = mac
        self.snmp_response = snmp_response
    def __repr__(self):
        return "[Host Record('%s','%s','%s')]" % (self.ip,
                                                  self.mac,
                                                  self.snmp_response)

def f(i,q,oq):
    while True:
        time.sleep(.1)
        if q.empty():
            sys.exit()
            print "Process Number: %s Exit" % i
        ip = q.get()
        print "Process Number: %s" % i
        ret = subprocess.call("ping -c 1 %s" % ip,
                              shell=True,
                              stdout=open('/dev/null', 'w'),
                              stderr=subprocess.STDOUT)
        if ret == 0:
            print "%s: is alive" % ip
            oq.put(ip)
```

```
        else:
          print "Process Number: %s didn't find a response for %s " % (i, ip)
           pass

def snmp_query(i,out):
    while True:
        time.sleep(.1)
        if out.empty():
            sys.exit()
            print "Process Number: %s" % i
        ipaddr = out.get()
        s = Snmp()
        h = HostRecord()
        h.ip = ipaddr
        h.snmp_response = s.query()
        print h
        return h
try:
    q.putmany(ips)
finally:
    for i in range(num_workers):
        p = Process(target=f, args=[i,q,oq])
        p.start()
    for i in range(num_workers):
        pp = Process(target=snmp_query, args=[i,oq])
        pp.start()

print "main process joins on queue"
p.join()
#while not oq.empty():
#    print "Validated", oq.get()

print "Main Program finished"
```

Когда мы запустили этот сценарий, то получили следующий результат:

```
[root@giftcsllc02][H:4849][J:0]> python discover.py
Process Number: 0
192.19.101.250: is alive
Process Number: 1
192.19.101.251: is alive
Process Number: 2
Process Number: 3
Process Number: 4
main process joins on queue
192.19.101.252: is alive
192.19.101.253: is alive
Main Program finished
[Host Record('192.19.101.250','None','('Linux linux.host 2.6.18-8.1.15.el5
  #1 SMP Mon Oct 22 08:32:04 EDT 2007 i686',)')]
[Host Record('192.19.101.252','None','('Linux linux.host 2.6.18-8.1.15.el5
  #1 SMP Mon Oct 22 08:32:04 EDT 2007 i686',)')]
[Host Record('192.19.101.253','None','('Linux linux.host 2.6.18-8.1.15.el5
```

```
  #1 SMP Mon Oct 22 08:32:04 EDT 2007 i686',)')]
[Host Record('192.19.101.251','None','('Linux linux.host 2.6.18-8.1.15.el5
  #1 SMP Mon Oct 22 08:32:04 EDT 2007 i686',)')]
Process Number: 4 didn't find a response for 192.168.1.1
```

Полученные результаты показывают, как работает этот интересный алгоритм исследования центра обработки данных. В этом сценарии можно было бы кое-что исправить: например, добавить запись MAC-адреса в объект `HostRecord`, переписать программный код в более объектно-ориентированном стиле, – дополнений хватило бы еще на одну книгу, и разработок хватило бы на целую компанию. Понимая это, мы переходим к другому разделу.

Получение множества значений с помощью SNMP

Получение единственного значения не представляет большой сложности, хотя иногда бывает желательно проверить ответы или выполнить некоторое действие, основанное, например, на типе операционной системы, под управлением которой работает компьютер. Чтобы сделать что-то более значимое, бывает необходимо получить несколько значений и выполнить какие-либо действия над ними.

Часто возникает задача произвести инвентаризацию центра обработки данных или отдела и собрать сведения о некоторых параметрах всех имеющихся машин. Представим такую гипотетическую ситуацию: вы готовитесь к крупному обновлению программного обеспечения и вам сказали, что каждая система должна иметь как минимум 1 Гбайт ОЗУ. Вы помните, что на большинстве компьютеров установлено памяти не менее 1 Гбайта, но среди тысяч поддерживаемых вами компьютеров имеется несколько таких, где памяти меньше требуемого объема.

Очевидно, что у вас имеется несколько вариантов действий. Рассмотрим каждый из них:

Вариант 1

Физически обойти все компьютеры и проверить объем ОЗУ на каждом из них, запустив некоторую команду или вскрыв корпус. Достаточно очевидно, что это не самый привлекательный вариант.

Вариант 2

Зайти по сети на каждый компьютер и выполнить команду, чтобы определить объем ОЗУ. Этот подход обладает массой недостатков, но его хотя бы теоретически можно реализовать в виде сценария, выполняющего команду средствами ssh. Одна из основных проблем состоит в том, что сценарий должен учитывать различия между платформами, так как все операционные системы в чем-то немного отличаются. Другая проблема заключается в необходимости знать, где все эти компьютеры располагаются.

Вариант 3

Написать небольшой сценарий, который опросит все устройства, включенные в сеть, и определит объем памяти на каждом из них с помощью SNMP.

Вариант 3, основанный на использовании SNMP, позволяет легко создать опись, в которой будут присутствовать только компьютеры, имеющие менее 1 Гбайта ОЗУ. Точный идентификатор OID, который потребуется запросить, носит имя «hrMemorySize». Протокол SNMP относится к разряду тех инструментов, которые всегда выгоднее использовать в многозадачном режиме, но все-таки подобной оптимизации лучше избегать, если это не является абсолютно необходимым. Помня об этом, перейдем непосредственно к деталям.

Чтобы быстро проверить нашу идею, воспользуемся программным кодом из предыдущего примера.

Получение объема памяти с помощью SNMP:

```
In [1]: run snmpinput

In [2]: who
netsnmp Snmp

In [3]: s = Snmp()

In [4]: s.DestHost = "10.0.1.2"

In [5]: s.Community = "public"

In [6]: s.oid = "hrMemorySize"

In [7]: result = int(s.query()[0])
   hrMemorySize = None ( None )

In [27]: print result
2026124
```

Как видите, реализовать подобный сценарий достаточно просто. Результат возвращается в виде кортежа в строке [6], поэтому мы извлекаем элемент с индексом 0 и преобразуем его в целое число. Теперь мы имеем целое число, соответствующее объему памяти в килобайтах. Единственное, что следует иметь в виду, – на разных компьютерах объем ОЗУ вычисляется по-разному. Поэтому в таких случаях лучше принимать решение на основе приближенных, а не точных значений. Например, можно было бы искать компьютеры, объем ОЗУ в которых немного меньше 1 Гбайта – скажем, 990 Мбайт.

В приведенном примере мы можем примерно оценить, что полученное число примерно соответствует объему ОЗУ 2 Гбайта. Вы можете полагаться на эту информацию, отвечая на запрос своего руководителя о наличии компьютеров с ОЗУ менее 2 Гбайт, если известно, что новое приложение, которое требуется установить, требует наличия памяти не менее 2 Гбайт.

Теперь, проверив основную идею, мы можем автоматизировать процедуру определения памяти. Если говорить более определенно, следует опросить все компьютеры, выяснить, в каких из них установлено памяти менее 2 Гбайт ОЗУ и затем записать полученную информацию в файл формата CSV[1], чтобы ее потом проще было импортировать в Excel или OpenOffice Calc.

Далее можно написать инструмент командной строки, который принимает диапазон сетевых адресов в качестве входного аргумента и, дополнительно, значение идентификатора OID, по умолчанию используя идентификатор «hrMemorySize». При этом в сценарии нам необходимо будет предусмотреть обход сетевых адресов из указанного диапазона.

Всякий раз, когда системный администратор пишет программный код, он сталкивается с определенными ограничениями. В состоянии ли вы потратить несколько часов или даже дней на создание большого объектно-ориентированного сценария, который потом можно будет использовать для решения других задач, или вам нужно быстро получить хотя бы приблизительные результаты? На наш взгляд, в большинстве случаев вполне можно выполнить обе реализации. При использовании IPython вы можете быстро создавать заготовки сценариев и затем доводить их до окончательного состояния. Вообще – это хорошая идея писать программный код многократного использования, поскольку эта привычка, как снежный ком, быстро обретает инерцию движения.

Надеемся, что теперь вы понимаете, в чем заключается сила SNMP. Давайте приступим к созданию нашего сценария...

Поиск объема памяти

В этом следующем примере мы реализуем инструмент командной строки, определяющий объем памяти, установленной в компьютерах, с помощью SNMP:

```
#!/usr/bin/env python
#Инструмент командной строки, определяющий общий объем памяти в компьютере

import netsnmp
import optparse
from IPy import IP

class SnmpSession(object):
    """Простой сеанс SNMP"""
    def __init__(self,
                 oid="hrMemorySize",
                 Version=2,
                 DestHost="localhost",
```

[1] CSV, или Comma Separated Values, – значения, разделенные запятыми. – *Прим. перев.*

```
                Community="public"):
        self.oid = oid
        self.Version = Version
        self.DestHost = DestHost
        self.Community = Community

    def query(self):
        """Создает запрос SNMP"""
        try:
            result = netsnmp.snmpwalk(self.oid,
                                      Version = self.Version,
                                      DestHost = self.DestHost,
                                      Community = self.Community)
        except:
            #Несмотря на то, что это всего лишь пример,
            #тем не менее, выведем, какое исключение возникло
            import sys
            print sys.exc_info()
            result = None
            return result

class SnmpController(object):
    """Использует модуль optparse для управления сеансом SnmpSession"""

    def run(self):
        results = {} #Место сбора и хранения результатов snmp
        p = optparse.OptionParser(description="A tool that determines
                                    memory installed",
                                prog="memorator",
                                version="memorator 0.1.0a",
                                usage="%prog [subnet range] [options]")
        p.add_option('--community', '-c',help='community string',
            default='public')
        p.add_option('--oid', '-o', help='object identifier',
            default='hrMemorySize')
        p.add_option('--verbose', '-v', action='store_true',
            help='increase verbosity')
        p.add_option('--quiet', '-q', action='store_true',help='
                    suppresses most messages')
        p.add_option('--threshold', '-t', action='store', type="int",
                    help='a number to filter queries with')

        options, arguments = p.parse_args()
        if arguments:
            for arg in arguments:
                try:
                    ips = IP(arg) #Преобразовать аргумент в строку
                except:
                    if not options.quiet:
                        print 'Ignoring %s, not a valid IP address' % arg
                    continue
                for i in ips:
                    ipAddr = str(i)
```

```
                        if not options.quiet:
                            print 'Running snmp query for: ', ipAddr
                        session = SnmpSession(options.oid,
                                              DestHost = ipAddr,
                                              Community = options.community)
                        if options.oid == "hrMemorySize":
                            try:
                                memory = int(session.query()[0])/1024
                            except:
                                memory = None
                            output = memory
                    else:
                        #Обработка результатов, не имеющих отношения
                        #к объему памяти
                        output = session.query()
                        if not options.quiet:
                            print "%s returns %s" % (ipAddr,output)

                    #Поместить полученные результаты в словарь,
                    #Но только если был получен корректный ответ
                    if output != None:
                        if options.threshold: #если порог обозначен
                            if output < options.threshold:
                                results[ipAddr] = output
                                #если разрешен вывод результатов
                                if not options.quiet:
                                    print "%s returns %s" % (ipAddr,output)
                        else:
                            results[ipAddr] = output
                            if not options.quiet:
                                print output

                print "Results from SNMP Query %s for %s:\n" % (options.oid,
                    arguments), results

            else:
                p.print_help() #при отсутствии аргументов командной строки
                               #вывести инструкцию об использовании
        def _main():
            """
                Запускает процесс сбора информации.
            """
            start = SnmpController()
            start.run()

    if __name__ =='__main__':
        try:
            import IPy
        except:
            print "Please install the IPy module to use this tool"
        _main()
```

Теперь пройдемся по этому сценарию и посмотрим, что он делает. Мы взяли целый класс из предыдущего примера и поместили его в новый модуль. Затем мы написали класс-контроллер, который анализирует аргументы командной строки с помощью модуля `optparse`. Модуль `IPy`, к которому мы обращаемся снова и снова, используется для автоматической обработки IP-адресов. Благодаря этому можно указать несколько IP-адресов или диапазон адресов, и наш модуль будет отсылать запросы SNMP и возвращать результаты в виде словаря, в котором роль ключей будут играть IP-адреса, а роль значений – ответы SNMP.

Единственная сложность, которая здесь реализована, – это логика обработки пустых ответов и проверки порогового значения. То есть модуль возвращает только значения ниже указанного порога. При использовании порога мы можем получать значимые для нас результаты и учесть различия в том, как разные компьютеры вычисляют объем памяти.

Посмотрим на вывод, полученный в результате работы этого модуля:

```
[ngift@ng-lep-lap][H:6518][J:0]> ./memory_tool_netsnmp.py 10.0.1.2 10.0.1.20
Running snmp query for: 10.0.1.2
    hrMemorySize = None ( None )
1978
Running snmp query for: 10.0.1.20
    hrMemorySize = None ( None )
372
Results from SNMP Query hrMemorySize for ['10.0.1.2', '10.0.1.20']:
{'10.0.1.2': 1978, '10.0.1.20': 372}
```

Как видите, результаты были получены для компьютеров в подсети 10.0.1.0/24. Теперь воспользуемся флагом threshold (порог), чтобы сымитировать поиск машин, объем ОЗУ в которых меньше 2 Гбайт. Как уже упоминалось выше, разные компьютеры по-разному вычисляют имеющийся объем ОЗУ, поэтому для пущей уверенности возьмем в качестве порогового значения число 1800, что примерно должно соответствовать объему ОЗУ 1800 Мбайт. То есть, если в компьютере объем ОЗУ составляет менее 1800 Мбайт, или примерно 2 Гбайта, информация о нем будет включена в наш отчет.

Ниже приводится результат выполнения такого запроса:

```
[ngift@ng-lep-lap][H:6519][J:0]>
    ./memory_tool_netsnmp.py --threshold 1800 10.0.1.2 10.0.1.20
Running snmp query for: 10.0.1.2
    hrMemorySize = None ( None )
Running snmp query for: 10.0.1.20
    hrMemorySize = None ( None )
10.0.1.20 returns 372
Results from SNMP Query hrMemorySize for ['10.0.1.2', '10.0.1.20']:
{'10.0.1.20': 372}
```

Наш сценарий прекрасно справился с заданием, однако мы можем сделать еще кое-что, чтобы оптимизировать его. Если вам потребуется опросить несколько тысяч машин, то для выполнения работы этому сценарию может потребоваться целый день или даже больше. Это, может быть, и не страшно, но если вам требуется получить результаты очень быстро, вам необходимо будет обеспечить возможность параллельного выполнения нескольких запросов одновременно и организовать ветвление для каждого запроса, используя для этого библиотеку стороннего производителя. Еще одно усовершенствование, которое можно было бы внести, – это автоматическое создание файла отчета в формате CSV из нашего словаря.

Но прежде чем мы перейдем к реализации этих задач, позвольте обратить ваше внимание на один момент, который вы, возможно, не заметили. Сценарий написан так, что позволяет запрашивать любой OID, а не только определять объем памяти. Это очень удобно, потому что у нас теперь имеется как инструмент определения объемов памяти, так и универсальный инструмент, позволяющий выполнять любые запросы SNMP.

Рассмотрим пример, который наглядно демонстрирует, что мы имеем в виду:

```
[ngift@ng-lep-lap][H:6522][J:0]> ./memory_tool_netsnmp.py -o sysDescr
10.0.1.2 10.0.1.20
Running snmp query for: 10.0.1.2
    sysDescr = None ( None )
10.0.1.2 returns ('Linux cent 2.6.18-8.1.14.el5 #1 SMP
 Thu Sep 27 19:05:32 EDT 2007 x86_64',)
('Linux cent 2.6.18-8.1.14.el5 #1 SMP Thu Sep 27 19:05:32 EDT 2007 x86_64',)
Running snmp query for: 10.0.1.20
    sysDescr = None ( None )
10.0.1.20 returns ('Linux localhost.localdomain 2.6.18-8.1.14.el5 #1 SMP
 Thu Sep 27 19:05:32 EDT 2007 x86_64',)
('Linux localhost.localdomain 2.6.18-8.1.14.el5 #1 SMP
 Thu Sep 27 19:05:32 EDT 2007 x86_64',)
Results from SNMP Query sysDescr for ['10.0.1.2', '10.0.1.20']:
{'10.0.1.2': ('Linux cent 2.6.18-8.1.14.el5 #1 SMP
Thu Sep 27 19:05:32 EDT 2007 x86_64',), '10.0.1.20':
('Linux localhost.localdomain 2.6.18-8.1.14.el5 #1 SMP
  Thu Sep 27 19:05:32 EDT 2007 x86_64',)}
```

Совсем не лишне учитывать это, приступая к работе над «одноразовым» инструментом для конкретного случая. Почему бы не потратить дополнительные 30 минут, чтобы придать ему универсальность? В результате у вас может получиться инструмент, который будет находить применение снова и снова, и эти 30 минут превратятся в ничто по сравнению с тем временем, которое вам удастся сэкономить в будущем.

Создание гибридных инструментов SNMP

Мы уже показали вам несколько отдельных инструментов и хотим отметить, что использованные нами приемы можно объединить для создания весьма сложных инструментов. Начнем с создания серии простых узкоспециализированных инструментов, на основе которых позднее мы сможем создавать большие сценарии.

Ниже приводится полезный сценарий с именем `snmpstatus`, который получает несколько различных запросов snmp и комбинирует из них «состояние» опрашиваемого узла:

```
import subprocess

class Snmpdf(object):
    """Инструмент командной строки snmpstatus"""
    def __init__(self,
                 Version="-v2c",
                 DestHost="localhost",
                 Community="public",
                 verbose=True):

        self.Version = Version
        self.DestHost = DestHost
        self.Community = Community
        self.verbose = verbose

    def query(self):
        """Создает запрос для snmpstatus"""
        Version = self.Version
        DestHost = self.DestHost
        Community = self.Community
        verbose = self.verbose

        try:
            snmpstatus = "snmpstatus %s -c %s %s" % (Version, Community,
                                                     DestHost)
            if verbose:
                print "Running: %s" % snmpstatus
            p = subprocess.Popen(snmpstatus,
                                 shell=True,
                                 stdout=subprocess.PIPE)
            out = p.stdout.read()
            return out

        except:
            import sys
            print >> sys.stderr, "error running %s" % snmpstatus

def _main():
    snmpstatus = Snmpdf()
    result = snmpstatus.query()
    print result
```

```
if __name__ == "__main__":
    _main()
```

Мы надеемся, что вы обратили внимание на тот факт, что этот сценарий не сильно отличается от команды `snmpdf`, за исключением некоторых имен. Это отличный пример, когда было бы желательно перейти на более высокий уровень абстракции и затем повторно использовать общие компоненты. Если бы мы создали модуль, вмещающий весь общий программный код, наш новый сценарий состоял бы всего из нескольких строк. Имейте это в виду, мы еще вернемся к этому.

Другой инструмент, имеющий отношение к SNMP, – это ARP, который использует протокол ARP. С помощью протокола ARP можно получить MAC-адреса устройств по их IP-адресам, при условии, что они находятся в одной и той же сети. Давайте напишем и этот узкоспециализированный инструмент. Он пригодится нам немного позже.

Оформить действия с протоколом ARP в виде сценария не составит никакого труда; можно сразу продемонстрировать работу этого примера, используя интерактивную оболочку IPython. Итак, запустите IPython и введите следующее:

```
import re
import subprocess

#некоторые переменные
ARP = "arp"
IP = "10.0.1.1"
CMD = "%s %s " % (ARP, IP)
macPattern = re.compile(":")

def getMac():
    p = subprocess.Popen(CMD, shell=True, stdout=subprocess.PIPE)
    out = p.stdout.read()
    results = out.split()
    for chunk in results:
        if re.search(macPattern, chunk):
            return chunk

if __name__ == "__main__":
    macAddr = getMac()
    print macAddr
```

Этот фрагмент нельзя назвать инструментом многократного использования, но вы легко можете взять эту идею за основу и использовать ее как часть общей библиотеки получения сведений об устройствах в сети центра обработки данных.

Расширение возможностей Net-SNMP

Как уже говорилось ранее, в большинстве операционных систем *nix пакет Net-SNMP установлен в виде агента. По умолчанию агент может возвращать определенный перечень информации, однако существует

возможность расширять этот перечень. Можно было бы указать агенту на необходимость собирать некоторые сведения и затем возвращать их по протоколу SNMP.

Файл *EXAMPLE.conf*, поставляемый в составе Net-SNMP, – это один из лучших источников информации по расширению возможностей Net-SNMP. Нелишним будет обратиться к команде `man snmpd.conf`, которая выводит более подробную информацию с описанием API. Если вас интересуют вопросы расширения возможностей «родных» агентов пакета, оба эти источника справочной информации могут стать для вас незаменимыми.

С точки зрения программистов на языке Python, возможность расширения Net-SNMP является одним из самых захватывающих аспектов работы с SNMP, потому что позволяет разработчикам писать программный код, выполняющий мониторинг всего, что они сочтут необходимым, и дополнительно иметь внутреннего агента, отвечающего предъявляемым условиям.

Пакет Net-SNMP предлагает достаточно много способов расширения возможностей агента, и для начала мы напишем программу «Hello World», которая будет выполняться по запросу `snmp`. Первый шаг заключается в создании простого файла *snmpd.conf*, посредством которого будет запускаться наша программа «Hello World», написанная на языке Python. В примере 7.4 показано, как выглядит этот файл в операционной системе Red Hat.

Пример 7.4. Конфигурационный файл SNMP, предусматривающий вызов программы «Hello World»

```
syslocation "O'Reilly"
syscontact bofh@oreilly.com
rocommunity public
exec helloworld /usr/bin/python -c "print 'hello world from Python'"
```

После этого следует сообщить демону `snmpd` о необходимости перечитать конфигурационный файл. Сделать это можно тремя разными способами. В Red Hat можно использовать такую команду:

```
service snmpd reload
```

или сначала выполнить такую команду:

```
ps -ef | grep snmpd
root 12345 1 0 Apr14 ?
00:00:30 /usr/sbin/snmpd -Lsd -Lf /dev/null -p /var/run/snmpd.pid -a
```

а затем послать демону сигнал:

```
kill -HUP 12345
```

Наконец, можно с помощью команды `snmpset` присвоить целое число (1) параметру `UCD-SNMPMIB::versionUpdateConfig.0` и тем самым вынудить демон `snmpd` перечитать конфигурационный файл.

Теперь, когда демон `snmpd` перечитал измененный файл *snmpd.conf*, мы можем двинуться дальше и послать нашей машине запрос с помощью команды `snmpwalk` или с помощью расширения Net-SNMP из оболочки IPython. Ниже показано, что возвращает команда `snmpwalk`:

```
[root@giftcsllc02][H:4904][J:0]> snmpwalk -v 1 -c public localhost
.1.3.6.1.4.1.2021.8
UCD-SNMP-MIB::extIndex.1 = INTEGER: 1
UCD-SNMP-MIB::extNames.1 = STRING: helloworld
UCD-SNMP-MIB::extCommand.1 = STRING: /usr/bin/python
  -c "print 'hello world from Python'"
UCD-SNMP-MIB::extResult.1 = INTEGER: 0
UCD-SNMP-MIB::extOutput.1 = STRING: hello world from Python
UCD-SNMP-MIB::extErrFix.1 = INTEGER: noError(0)
UCD-SNMP-MIB::extErrFixCmd.1 = STRING:
```

Этот запрос требует некоторых пояснений, так как наблюдательный читатель может задаться вопросом, откуда взялся OID 1.3.6.1.4.1.2021.8. Этот OID соответствует идентификатору `ucdavis.extTable`. Когда создается расширение в *snmpd.conf*, оно присваивается этому OID. Дело несколько осложняется, когда возникает потребность создать свой OID. Для этого необходимо обратиться в организацию *iana.org* и получить уникальный номер для своего предприятия. После этого можно будет использовать полученный номер для создания специализированных запросов агенту. Основная причина таких сложностей состоит в необходимости сохранить однородность пространства имен и избежать конфликтов с числами, которые, возможно, получат поставщики оборудования в будущем.

Истинная сила Python заключается вовсе не в том, чтобы получить вывод от единственной команды – это было бы слишком просто. Ниже приводится пример сценария, который определяет общее число обращений к веб-серверу Apache из броузера Firefox, анализируя файл журнала, и возвращает результат под нестандартным OID предприятия. Начнем рассмотрение с конца и сначала посмотрим на полученные результаты:

```
snmpwalk -v 2c -c public localhost .1.3.6.1.4.1.2021.28664.100
UCD-SNMP-MIB::ucdavis.28664.100.1.1 = INTEGER: 1
UCD-SNMP-MIB::ucdavis.28664.100.2.1 = STRING: "FirefoxHits"
UCD-SNMP-MIB::ucdavis.28664.100.3.1 = STRING:
 "/usr/bin/python /opt/local/snmp_scripts/agent_ext_logs.py"
UCD-SNMP-MIB::ucdavis.28664.100.100.1 = INTEGER: 0
UCD-SNMP-MIB::ucdavis.28664.100.101.1 = STRING:
  "Total number of Firefox Browser Hits: 15702"
UCD-SNMP-MIB::ucdavis.28664.100.102.1 = INTEGER: 0
UCD-SNMP-MIB::ucdavis.28664.100.103.1 = ""
```

Если отыскать строку со значением 100.101.1, можно увидеть вывод, полученный от сценария, который анализирует файл журнала веб-сервера Apache и отыскивает записи, свидетельствующие об обращениях

с помощью броузера Firefox. Затем сценарий суммирует их и возвращает по протоколу SNMP. В примере 7.5 приводится исходный текст сценария, который запускается при выполнении запроса к данному OID.

Пример 7.5. Сценарий поиска числа обращений к веб-серверу Apache из броузера Firefox

```
import re

"""Возвращает число обращений из броузера Firefox"""

def grep(lines,pattern="Firefox"):
    pat = re.compile(pattern)
    for line in lines:
        if pat.search(line): yield line

def increment(lines):
    num = 0
    for line in lines:
        num += 1
    return num

wwwlog = open("/home/noahgift/logs/noahgift.com-combined-log")
column = (line.rsplit(None,1)[1] for line in wwwlog)
match = grep(column)
count = increment(match)
print "Total Number of Firefox Hits: %s" % count
```

Чтобы заставить этот запрос работать, мы сначала должны добавить в файл *snmpd.conf* информацию об этом сценарии, как показано ниже:

```
syslocation "O Reilly"
syscontact bofh@oreilly.com
rocommunity public
exec helloworld /usr/bin/python -c "print 'hello world from Python'"
exec .1.3.6.1.4.1.2021.28664.100 FirefoxHits /usr/bin/python
  /opt/local/snmp_scripts/agent_ext_logs.py
```

Самая магическая часть здесь – последняя строка с идентификатором .1.3.6.1.4.1.2021, где 28664 является числом нашего предприятия, а число 100 – просто некоторое число, которое мы решили использовать для примера. Это очень важно – следовать общепринятым правилам и использовать свое число предприятия, если вы планируете заниматься расширением возможностей SNMP. Благодаря этому вы сумеете избежать конфликтов при использовании в команде `snmpset` чисел, уже занятых кем-то другим.

Мы склонны считать, что тема использования SNMP является одной из самых захватывающих тем в книге и что при этом SNMP по-прежнему остается малоизведанной областью. Можно привести массу примеров, когда расширение возможностей Net-SNMP может быть полезным, а при аккуратном использовании SNMPv3 вы сможете делать удивительные вещи, реализовать которые с помощью протокола SNMP

совсем несложно и для которых применение ssh и сокетов могло бы показаться естественным выбором.

Управление устройствами через SNMP

Одним из самых интересных аспектов применения SNMP является возможность управления устройствами по этому протоколу. Очевидно, что такой способ управления маршрутизатором обладает существенными преимуществами перед использованием, например, модуля Pexpect (*http://sourceforge.net/projects/pexpect/*), потому что реализуется намного проще.

Для краткости мы в примере будем рассматривать только использование SNMPv1, но, если вам предстоит взаимодействовать с устройствами через незащищенную сеть, вам следует использовать SNMPv3. Перед прочтением этого раздела было бы неплохо ознакомиться с книгами «Essential SNMP» и «Cisco IOS Cookbook» Кевина Дули (Kevin Dooley) и Яна Дж. Брауна (Ian. J. Brown) (O'Reilly), если они у вас имеются или у вас имеется учетная запись для доступа к службе Safari. Они содержат обширую информацию как об основах настройки, так и о способах взаимодействия с устройствами Cisco по протоколу SNMP.

Поскольку перезагрузка параметров настройки в устройствах Cisco красиво реализуется через протокол SNMP, мы выбрали эту тему для разговора об управлении устройствами. Для опробования этого примера вам потребуется работающий сервер TFTP, откуда маршрутизатор будет забирать файл IOS, и маршрутизатор с разрешенным доступом для чтения/записи по протоколу SNMP. В примере 7.6 приводится сценарий на языке Python.

Пример 7.6. Выгрузка новой конфигурации в маршрутизатор Cisco

```
import netsnmp

vars = netsnmp.Varbind(netsnmp.VarList(netsnmp.Varbind(
                                          ".1.2.6.1.4.1.9.2.10.6.0", "1"),
    (netsnmp.Varbind("cisco.example.com.1.3.6.1.4.1.9.2.10.12.172.25.1.1",
                     "iso-config.bin")

result = netsnmp.snmpset(vars,
                         Version = 1,
                         DestHost='cisco.example.com',
                         Community='readWrite')
```

В этом примере мы использовали метод `VarList` из модуля `netsnmp`, чтобы сначала выполнить инструкцию, которая стирает информацию во флеш-памяти коммутатора, а затем загрузить новый образ файла IOS. Этот программный код мог бы послужить основой сценария, выполняющего обновление настроек всех коммутаторов в вычислительном центре. Как и любой другой программный код в этой книге, он должен

быть опробован на оборудовании, не включенном в работу, и вы не окажетесь перед фактом, что что-то натворили,.

И последнее замечание: протокол SNMP редко рассматривается как способ управления устройствами и, тем не менее, он предоставляет широкие возможности по управлению устройствами в вычислительном центре, поскольку является универсальной спецификацией для устройств, выпускавшихся начиная с 1988 года. В будущем возможно очень интересное развитие протокола SNMP v3.

Интеграция SNMP в сеть предприятия с помощью Zenoss

Zenoss представляет собой замечательную систему управления локальными сетями уровня предприятия. Мало того, что Zenoss является приложением, распространяемым с открытыми исходными текстами, но оно еще целиком написано на языке Python. Система Zenoss является представителем нового поколения приложений уровня предприятия, обладающих большими возможностями и допускающих расширение с использованием интерфейса XML-RPC или ReST. За дополнительной информацией о ReST обращайтесь к книге Леонарда Ричардсона (Leonard Richardson) и Сэма Руби (Sam Ruby) «RESTful Web Services» (O'Reilly).

Наконец, если у вас появится желание участвовать в разработке Zenoss, вы можете предлагать свои исправления.

Прикладной интерфейс Zenoss

За последней информацией о прикладном интерфейсе Zenoss обращайтесь на сайт *http://www.zenoss.com/community/docs/howtos/send-events/*.

Использование Zendmd

Система Zendoss не только поставляется в комплекте с системой мониторинга и исследования SNMP, но и включает в себя прикладной интерфейс высокого уровня с именем zendmd. Вы можете открыть настроенную командную оболочку Python и выполнять команды Zenoss непосредственно.

Пример использования zendmd:

```
>>> d = find('build.zenoss.loc')
>>> d.os.interfaces.objectIds()
['eth0', 'eth1', 'lo', 'sit0', 'vmnet1', 'vmnet8']
>>> for d in dmd.Devices.getSubDevices():
>>>     print d.id, d.getManageIp()
```

Прикладной интерфейс доступа к устройствам

С системой Zenoss можно также взаимодействовать через интерфейс XML-RPC и добавлять или удалять устройства. Ниже приводятся два примера:

С использованием ReST:

```
[zenos@zenoss $]
wget 'http://admin:zenoss@MYHOST:8080/zport/dmd
/ZenEventManager/manage_addEvent?device=MYDEVICE&component=¬
MYCOMPONENT&summary=¬
MYSUMMARY&severity=4&eclass=EVENTCLASS&eventClassKey=EVENTCLASSKEY
```

С использованием XML-RPC:

```
>>> from xmlrpclib import ServerProxy
>>> serv = ServerProxy(
                'http://admin:zenoss@MYHOST:8080/zport/dmd/ZenEventManager')
>>> evt = {'device':'mydevice', 'component':'eth0',
           'summary':'eth0 is down','severity':4, 'eventClass':'/Net'}
>>> serv.sendEvent(evt)
```

8

Окрошка из операционных систем

Введение

Быть системным администратором зачастую означает быть брошенным на съедение волкам. Правила, предварительное планирование и даже выбор операционной системы часто находятся вне сферы вашего влияния. В настоящее время чтобы быть хотя бы мало-мальски эффективным системным администратором, вам необходимо знать их все, мы имеем в виду все операционные системы. От Linux до Solaris, Mac OS X и FreeBSD – все эти системы должны быть вам знакомы. Только время покажет, продолжат ли свое существование такие патентованные операционные системы, как AIX или HP-UX, но они все еще необходимы многим людям.

К счастью, здесь нам на помощь опять приходит язык Python – мы надеемся, что вы обратили внимание, что в состав языка входит полномасштабная стандартная библиотека, которая способна удовлетворить практически все потребности администраторов самых разнообразных операционных систем. В составе стандартной библиотеки имеются модули, которые позволят системному администратору реализовать все, что ему необходимо, от архивирования каталогов и сравнения файлов и каталогов до анализа конфигурационных файлов. Зрелость языка Python вместе с его элегантностью и удобочитаемостью – причина того, что он не имеет себе равных в системном администрировании.

Во многих сложнейших областях человеческой деятельности, где требуются услуги системного администратора, таких как производство мультипликационных фильмов или в вычислительных центрах, происходит отказ от применения языка Perl в пользу языка Python, потому что последний позволяет писать более элегантный и удобочитаемый программный код. Язык Ruby – это достаточно интересный язык программирования, в котором используются положительные особен-

ности языка Python, но, тем не менее, мощность стандартной библиотеки и широта возможностей языка Python дает ему преимущества перед языком Ruby при использовании в качестве языка системного администрирования.

В этой главе будут рассматриваться несколько операционных систем, поэтому у нас не будет времени исследовать какую-либо из них достаточно глубоко, но мы углубимся настолько, чтобы показать, что Python может играть роль как универсального кросс-платформенного языка сценариев, так и уникального средства администрирования каждой из операционных систем. Кроме того, на горизонте замаячила «новая операционная система», которая обретает форму центра обработки данных. Эта новая платформа получила название «(за)облачная» обработка данных (Cloud Computing), и мы поговорим о том, что предлагают компании Amazon и Google.

Но довольно бездельничать и балагурить. С кухни потянуло чем-то восхитительным... это что, окрошка из операционных систем?

Кросс-платформенное программирование на языке Python в UNIX

Хотя между разными UNIX-подобными операционными системами существуют некоторые значимые различия, но общего в них намного больше. Один из способов примирить различные версии *nix состоит в том, чтобы создавать кросс-платформенные инструменты и библиотеки, которые скрывают различия между операционными системами. Основной способ добиться этого – использовать условные инструкции, которые проверяют тип и версию операционной системы.

Язык Python неизменно следует философии «батарейки входят в комплект поставки» и предоставляет инструменты для решения практически любой проблемы, с которой вы можете столкнуться. Для определения типа платформы, на которой выполняется ваш программный код, существует модуль `platform`. Давайте поближе познакомимся с основами использования этого модуля.

Самый простой способ познакомиться с возможностями модуля `platform` – написать сценарий, который будет выводить всю доступную информацию о системе, как показано в примере 8.1.

Пример 8.1. Использование модуля platform для получения информации о системе

```
#!/usr/bin/env python
import platform

profile = [
platform.architecture(),
platform.dist(),
```

```
platform.libc_ver(),
platform.mac_ver(),
platform.machine(),
platform.node(),
platform.platform(),
platform.processor(),
platform.python_build(),
platform.python_compiler(),
platform.python_version(),
platform.system(),
platform.uname(),
platform.version(),
]

for item in profile:
    print item
```

Ниже приводится результат работы этого сценария в операционной системе OS X Leopard 10.5.2:

```
[ngift@Macintosh-6][H:10879][J:0]% python cross_platform.py
('32bit', '')
('', '', '')
('', '')
('10.5.2', ('', '', ''), 'i386')
i386
Macintosh-6.local
Darwin-9.2.0-i386-32bit
i386
('r251:54863', 'Jan 17 2008 19:35:17')
GCC 4.0.1 (Apple Inc. build 5465)
2.5.1
Darwin
('Darwin', 'Macintosh-6.local', '9.2.0', 'Darwin Kernel Version 9.2.0:
Tue Feb 5 16:13:22 PST 2008; root:xnu-1228.3.13~1/
RELEASE_I386','i386','i386')
Darwin Kernel Version 9.2.0: Tue Feb 5 16:13:22 PST 2008;
root:xnu-1228.3.13~1/RELEASE_I386
```

Этот пример позволяет получить представление о том, какого рода информация об операционной системе нам доступна. Следующий шаг на пути к созданию кросс-платформенного программного кода состоит в необходимости создать модуль `fingerprint`, который будет «брать отпечатки пальцев», определяя, на какой платформе, с каким номером версии он выполняется. В следующем примере мы «взяли отпечатки пальцев» у следующих операционных систем: Mac OS X, Ubuntu, Red Hat/CentOS, FreeBSD и SunOS. Взгляните на пример 8.2.

Пример 8.2. Определение типа операционной системы

```
#!/usr/bin/env python
import platform
```

```
"""
Определение принадлежности к одной из следующих операционных систем:
* Mac OS X
* Ubuntu
* Red Hat/Cent OS
* FreeBSD
* SunOS
"""
class OpSysType(object):
    """Определяет тип ОС с помощью модуля platform"""
    def __getattr__(self, attr):
        if attr == "osx":
            return "osx"
        elif attr == "rhel":
            return "redhat"
        elif attr == "ubu":
            return "ubuntu"
        elif attr == "fbsd":
            return "FreeBSD"
        elif attr == "sun":
            return "SunOS"
        elif attr == "unknown_linux":
            return "unknown_linux"
        elif attr == "unknown":
            return "unknown"
        else:
            raise AttributeError, attr

    def linuxType(self):
        """Определяет разновидность Linux с помощью различных методов"""
        if platform.dist()[0] == self.rhel:
            return self.rhel
        elif platform.uname()[1] == self.ubu:
            return self.ubu
        else:
            return self.unknown_linux

    def queryOS(self):
        if platform.system() == "Darwin":
            return self.osx
        elif platform.system() == "Linux":
            return self.linuxType()
        elif platform.system() == self.sun:
            return self.sun
        elif platform.system() == self.fbsd:
            return self.fbsd

    def fingerprint():
        type = OpSysType()
        print type.queryOS()

if __name__ == "__main__":
    fingerprint()
```

Теперь посмотрим, что выводит этот модуль при запуске на различных платформах.

Red Hat:

```
[root@localhost]/# python fingerprint.py
redhat
```

Ubuntu:

```
root@ubuntu:/# python fingerprint.py
ubuntu
```

Solaris 10 или SunOS:

```
bash-3.00# python fingerprint.py
SunOS
```

FreeBSD:

```
# python fingerprint.py
FreeBSD
```

Хотя в выводе этой команды не содержится ничего особенно интересного, но в действительности она предоставляет нам очень мощный инструмент. Этот простой модуль позволит нам писать кросс-платформенный программный код, так как мы, например, можем определить словарь с типами этих операционных систем и при нахождении соответствия выполнять соответствующий платформозависимый программный код. Одним из примеров получения самой ощутимой выгоды от использования приемов кросс-платформенного программирования могут служить сценарии, используемые для администрирования сети посредством применения ssh с ключами. В этом случае программный код может работать на многих платформах и давать непротиворечивые результаты.

Использование SSH с ключами, каталога NFS и кросс–платформенных сценариев Python для управления системами

Один из способов управления инфраструктурой из компьютеров, работающих под управлением разнотипных систем *nix, заключается в использовании ssh с ключами, общего каталога, монтируемого как том NFS, и кросс-платформенного программного кода на языке Python. Разобьем этот процесс на несколько шагов, чтобы было понятнее:

Шаг 1: создать открытый ключ ssh в системе, откуда будет выполняться администрирование. Обратите внимание: для разных платформ эта процедура может существенно отличаться. За подробностями обращайтесь к документации по операционной системе и к справочному руководству по команде ssh. Создание ключа демонстрируется в примере 8.3.

Примечание к примеру ниже: с целью демонстрации мы создали ключ для пользователя root, однако для обеспечения более высокого уровня безопасности было бы лучше создать учетную запись пользователя, для которого определить привилегии sudo на запуск только этого сценария.

Пример 8.3. Создание открытого ключа ssh

```
[ngift@Macintosh-6][H:11026][J:0]% ssh-keygen -t rsa
Generating public/private rsa key pair.
Enter file in which to save the key (/root/.ssh/id_rsa):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /root/.ssh/id_rsa.
Your public key has been saved in /root/.ssh/id_rsa.pub.
The key fingerprint is:
6c:2f:6e:f6:b7:b8:4d:17:05:99:67:26:1c:b9:74:11 root@localhost.localdomain
[ngift@Macintosh-6][H:11026][J:0]%
```

Шаг 2: Скопировать открытый ключ на администрируемые машины и создать файл *authorized_keys*, как показано в примере 8.4.

Пример 8.4. Копирование открытого ключа ssh

```
[ngift@Macintosh-6][H:11026][J:0]% scp id_leop_lap.pub root@10.0.1.51:~/
.ssh/
root@10.0.1.51's password:
id_leop_lap.pub
100% 403 0.4KB/s 00:00
[ngift@Macintosh-6][H:11027][J:0]% ssh root@10.0.1.51
root@10.0.1.51's password:
Last login: Sun Mar 2 06:26:10 2008
[root@localhost]~# cd .ssh
[root@localhost]~/.ssh# ll
total 8
-rw-r--r-- 1 root root 403 Mar 2 06:32 id_leop_lap.pub
-rw-r--r-- 1 root root 2044 Feb 14 05:33 known_hosts
[root@localhost]~/.ssh# cat id_leop_lap.pub > authorized_keys
[root@localhost]~/.ssh# exit

Connection to 10.0.1.51 closed.
[ngift@Macintosh-6][H:11028][J:0]% ssh root@10.0.1.51
Last login: Sun Mar 2 06:32:22 2008 from 10.0.1.3
[root@localhost]~#
```

Шаг 3: смонтировать общий каталог NFS, содержащий модули, которые потребуется запускать на стороне клиентов. Часто самый простой способ добиться этого заключается в использовании функции autofs и в последующем создании символической ссылки. Однако то же самое можно реализовать на основе системы управления версиями, с помощью которой через ssh обновлять локальные репозитарии SVN с программным кодом на администрируемых машинах. После таких

обновлений сценарии будут использовать самые свежие версии модулей. Например, в системе на базе дистрибутива Red Hat можно было бы выполнить, например, такую команду:

```
ln -s /net/nas/python/src /src
```

Шаг 4: написать сценарий, который будет запускать программный код на удаленных машинах. Теперь, когда у нас имеются ключи ssh и смонтированный каталог NFS (или каталог, находящийся под контролем системы управления версиями), это достаточно простая задача. Как обычно, начнем с примера наиболее простого сценария, выполняющего администрирование удаленных систем через ssh. Если ранее вам никогда не приходилось делать ничего подобного, вас наверняка удивит, насколько просто можно выполнять достаточно сложные действия. В примере 8.5 реализован запуск простой команды `uname`.

Пример 8.5. Простой управляющий сценарий

```
#!/usr/bin/env python
import subprocess

"""
Система управления на основе ssh
"""
machines = ["10.0.1.40",
"10.0.1.50",
"10.0.1.51",
"10.0.1.60",
"10.0.1.80"]

cmd = "uname"
for machine in machines:
    subprocess.call("ssh root@%s %s" % (machine, cmd), shell=True)
```

Выполнив этот сценарий на пяти машинах с указанными IP-адресами, которые работают под управлением CentOS 5, FreeBSD 7, Ubuntu 7.1 и Solaris 10, мы получили следующие результаты:

```
[ngift@Macintosh-6][H:11088][J:0]% python dispatch.py
Linux
Linux
Linux
SunOS
FreeBSD
```

Однако у нас имеется модуль, более точно определяющий тип операционной системы, поэтому используем его для получения более точного описания машин, которым посылаются команды, для чего создадим временный каталог *src* на каждой удаленной машине и скопируем туда наш программный код. Конечно, после создания управляющего сценария становится очевидной потребность в устойчивом интерфейсе командной строки к нему, так как в противном случае нам придется

изменять сам сценарий, чтобы выполнить какую-нибудь другую команду, как показано ниже:

```
cmd = "mkdir /src"
```

или:

```
cmd = "python /src/fingerprint.py"
```

или даже:

```
subprocess.call("scp fingerprint.py root@%s:/src/" % machine, shell=True)
```

Мы сделаем это, как только запустим наш сценарий `fingerprint.py`, но сначала посмотрим на новую команду:

```
#!/usr/bin/env python
import subprocess
"""
Система управления на основе ssh
"""
machines = ["10.0.1.40",
"10.0.1.50",
"10.0.1.51",
"10.0.1.60",
"10.0.1.80"]

cmd = "python /src/fingerprint.py"
for machine in machines:
    subprocess.call("ssh root@%s %s" % (machine, cmd), shell=True)
```

А теперь посмотрим, что получилось:

```
[ngift@Macintosh-6][H:11107][J:0]# python dispatch.py
redhat
ubuntu
redhat
SunOS
FreeBSD
```

Благодаря модулю `fingerprint.py` результаты выглядят намного лучше. Безусловно, несколько строк в нашем управляющем сценарии требуют кардинальной перестройки, потому что в противном случае нам всякий раз будет необходимо редактировать его. Нам требуется более удобный инструмент, поэтому давайте создадим его.

Создание кросс-платформенного инструмента управления

Решение об использовании ключей ssh в соединении с простой системой управления оказалось недостаточно удобным, потому что его сложно расширять или повторно использовать. Попробуем определить перечень проблем, характерных для предыдущего инструмента, а затем

составим список требований, устраняющих эти проблемы. Проблемы: список администрируемых машин определяется жестко, в самом сценарии; выполняемая команда жестко задана в сценарии; допускается запуск только одной команды за раз; один и тот же набор команд выполняется на всех машинах, мы лишены возможности выбора; наш управляющий сценарий ожидает, пока не будет получен ответ на каждую команду. Требования: нам необходим инструмент командной строки, который будет получать IP-адреса и команды, которые надлежит выполнить, из конфигурационного файла; нам необходим интерфейс командной строки с параметрами, чтобы можно было передавать команды машинам; нам необходим инструмент управления, который будет запускать команды в отдельных потоках выполнения, чтобы не блокировать процесс.

Похоже, что нам необходимо выработать элементарный синтаксис конфигурационного файла с разделом для машин и с разделом для команд. Взгляните на пример 8.6.

Пример 8.6. Конфигурационный файл для управляющего сценария

```
[MACHINES]
CENTOS: 10.0.1.40
UBUNTU: 10.0.1.50
REDHAT: 10.0.1.51
SUN: 10.0.1.60
FREEBSD: 10.0.1.80
[COMMANDS]
FINGERPRINT : python /src/fingerprint.py
```

Теперь нам необходимо написать функцию, которая будет читать содержимое конфигурационного файла и выделять разделы MACHINES и COMMANDS, чтобы можно было выполнять обход этих разделов по очереди, как показано в примере 8.7.

Следует заметить, что команды из конфигурационного файла будут импортироваться в случайном порядке. В большинстве случаев это может оказаться неприемлемым и, возможно, было бы лучше просто написать модуль на языке Python, который будет играть роль конфигурационного файла.

Пример 8.7. Улучшенный сценарий управления

```
#!/usr/bin/env python
import subprocess
import ConfigParser
…
Система управления на основе ssh
"""
def readConfig(file="config.ini"):
    """
    Извлекает IP-адреса и команды из конфигурационного файла
```

```
    и возвращает кортеж
    """
    ips = []
    cmds = []
    Config = ConfigParser.ConfigParser()
    Config.read(file)
    machines = Config.items("MACHINES")
    commands = Config.items("COMMANDS")
    for ip in machines:
        ips.append(ip[1])
    for cmd in commands:
        cmds.append(cmd[1])
    return ips, cmds

ips, cmds = readConfig()

#Выполнить все команды для каждого IP-адреса
for ip in ips:
    for cmd in cmds:
        subprocess.call("ssh root@%s %s" % (ip, cmd), shell=True)
```

Эти несложные изменения повысили удобство использования. Мы можем произвольно изменять список команд и машин и выполнять сразу все команды. Если теперь взглянуть на вывод сценария, можно убедиться, что он не изменился:

```
[ngift@Macintosh-6][H:11285][J:0]# python advanced_dispatch1.py
redhat
redhat
ubuntu
SunOS
FreeBSD
```

Хотя это весьма усовершенствованный инструмент, у нас по-прежнему отсутствует механизм выполнения команд в отдельных потоках, наличие которого определено в нашей спецификации. К счастью, мы можем воспользоваться некоторыми приемами, описанными в главе, посвященной процессам, и легко реализовать многопоточный режим выполнения. В примере 8.8 показано, что для этого можно сделать.

Пример 8.8. Многопоточный инструмент управления командами

```
#!/usr/bin/env python
import subprocess
import ConfigParser
from threading import Thread
from Queue import Queue
import time
"""
Многопоточная система управления на основе ssh
"""
start = time.time()
queue = Queue()
```

```
def readConfig(file="config.ini"):
    """
    Извлекает IP-адреса и команды из конфигурационного файла
    и возвращает кортеж
    """
    ips = []
    cmds = []
    Config = ConfigParser.ConfigParser()
    Config.read(file)
    machines = Config.items("MACHINES")
    commands = Config.items("COMMANDS")
    for ip in machines:
        ips.append(ip[1])
    for cmd in commands:
        cmds.append(cmd[1])
    return ips, cmds

def launcher(i,q, cmd):
    """Запускает команды в потоке выполнения, отдельном для каждого IP"""
    while True:
        #Получить ip, cmd из очереди
        ip = q.get()
        print "Thread %s: Running %s to %s" % (i, cmd, ip)
        subprocess.call("ssh root@%s %s" % (ip, cmd), shell=True)
        q.task_done()

#Получить IP-адреса и команды из конфигурационного файла
ips, cmds = readConfig()

#Определить количество используемых потоков, но не более 25
if len(ips) < 25:
    num_threads = len(ips)
else:
    num_threads = 25

#Запустить потоки
for i in range(num_threads):
    for cmd in cmds:
        worker = Thread(target=launcher, args=(i, queue,cmd))
        worker.setDaemon(True)
        worker.start()

print "Main Thread Waiting"
for ip in ips:
    queue.put(ip)
queue.join()
end = time.time()
print "Dispatch Completed in %s seconds" % end - start
```

Если теперь взглянуть на вывод нового многопоточного механизма управления, можно заметить, что на выполнение всех команд потребовалось около 1,2 секунды. Чтобы увидеть различия в скорости выпол-

нения, нам следует добавить измерение времени в оригинальный управляющий сценарий и сравнить полученные результаты:

```
[ngift@Macintosh-6][H:11296][J:0]# python threaded_dispatch.py
Main Thread Waiting
Thread 1: Running python /src/fingerprint.py to 10.0.1.51
Thread 2: Running python /src/fingerprint.py to 10.0.1.40
Thread 0: Running python /src/fingerprint.py to 10.0.1.50
Thread 4: Running python /src/fingerprint.py to 10.0.1.60
Thread 3: Running python /src/fingerprint.py to 10.0.1.80
redhat
redhat
ubuntu
SunOS
FreeBSD
Dispatch Completed in 1 seconds
```

После добавления в оригинальный управляющий сценарий простого программного кода, выполняющего замер времени, мы получили следующее:

```
[ngift@Macintosh-6][H:11305][J:0]# python advanced_dispatch1.py
redhat
redhat
ubuntu
SunOS
FreeBSD
Dispatch Completed in 3 seconds
```

Исходя из этих результатов, можно сказать, что наша многопоточная версия оказалась примерно в три раза быстрее. Если бы мы использовали этот инструмент для опроса сети, скажем, из 500 машин, а не из 5, разница в производительности могла бы оказаться еще более существенной. Пока разработка нашего кросс-платформенного инструмента управления продвигается достаточно успешно, поэтому сделаем следующий шаг и создадим кросс-платформенный инструмент сборки.

Следует заметить, что для реализации этого сценария, возможно, более удачным решением было бы использование многозадачной версии IPython. За подробностями обращайтесь по адресу: *http://ipython.scipy.org/moin/Parallel_Computing*.

Создание кросс-платформенного инструмента сборки

Мы уже знаем, как распределять работу между несколькими машинами, как определять тип операционной системы, под управлением которой выполняется сценарий, и, наконец, создавать универсальную декларацию с помощью менеджера пакетов EPM, который способен создавать специализированные пакеты в зависимости от типа операционной системы. Почему бы нам не объединить все это вместе? Мы

можем использовать эти три приема, чтобы создать кросс-платформенный инструмент сборки.

С появлением технологии виртуальных машин стало весьма просто создавать виртуальные машины для любых свободно распространяемых UNIX-подобных операционных систем, таких как Debian/Ubuntu, Red Hat/CentOS, FreeBSD и Solaris. Теперь, создав инструмент, который вы хотите сделать доступным всему миру или просто коллегам в вашей компании, вы легко и просто можете создать «сборочный цех», возможно даже на своем ноутбуке, где ведется разработка сценария, и создавать специализированные пакеты сразу же с этим сценарием.

Как это будет работать? Самый простой способ достичь этого – создать общее дерево сборки пакета на разделе NFS и предоставить доступ к этой точке монтирования всем серверам сборки. После этого, используя инструменты, созданные нами ранее, настроить серверы на сборку пакетов в каталоге NFS. Менеджер пакетов EPM позволяет создавать простые декларации, или «списки» файлов, кроме того, у нас имеется модуль `fingerprint`, поэтому самое сложное уже позади. Осталось написать программный код, который делает только то, что осталось.

Ниже показано, как может выглядеть сценарий сборки:

```
#!/usr/bin/env python
from fingerprint import fingerprint
from subprocess import call

os = fingerprint()

#Получить корректный ключ для EPM
epm_keyword = {"ubuntu":"dpkg", "redhat":"rpm", "SunOS":"pkg", "osx":"osx"}

try:
    epm_keyword[os]
except Exception, err:
    print err

subprocess.call("epm -f %s helloEPM hello_epm.list" %
platform_cmd,shell=True)
```

Теперь можно отредактировать конфигурационный файл *config.ini*, переориентировав его на запуск нового сценария.

```
[MACHINES]
CENTOS: 10.0.1.40
UBUNTU: 10.0.1.50
REDHAT: 10.0.1.51
SUN: 10.0.1.60
FREEBSD: 10.0.1.80
[COMMANDS]
FINGERPRINT = python /src/create_package.py
```

Теперь осталось только запустить многопоточную версию инструмента сборки и – эврика! – нам удалось создать пакеты для CentOS, Ubuntu,

Red Hat, FreeBSD и Solaris за считанные секунды. Этот сценарий еще нельзя рассматривать как окончательную рабочую версию программного кода, так как в нем отсутствует обработка ошибок, но он прекрасно демонстрирует, что позволяет сделать язык Python на скорую руку, всего за несколько минут или часов.

PyInotify

Если вам выпадет честь работать с платформами GNU/Linux, вы по достоинству оцените возможности PyInotify. Согласно документации это: «модуль Python для обнаружения изменений в файловой системе». Официальная страница проекта находится по адресу *http://pyinotify.sourceforge.net*.

В примере 8.9 показано, как работать с этим модулем.

Пример 8.9. Сценарий слежения за событиями с помощью модуля Pyinotify

```
import os
import sys
from pyinotify import WatchManager, Notifier, ProcessEvent, EventsCodes

class PClose(ProcessEvent):
    """
    Обработка события закрытия
    """
    def __init__(self, path):
        self.path = path
        self.file = file

    def process_IN_CLOSE(self, event):
        """
        Обработка событий 'IN_CLOSE_*'
        может принимать функцию-обработчик
        """
        path = self.path
        if event.name:
            self.file = "%s" % os.path.join(event.path, event.name)
        else:
            self.file = "%s" % event.path
        print "%s Closed" % self.file
        print "Performing pretend action on %s...." % self.file
        import time
        time.sleep(2)
        print "%s has been processed" % self.file

class Controller(object):

    def __init__(self, path='/tmp'):
        self.path = path

    def run(self):
        self.pclose = PClose(self.path)
```

```
            PC = self.pclose
            # следить только за этими событиями
            mask = EventsCodes.IN_CLOSE_WRITE | EventsCodes.IN_CLOSE_NOWRITE

            # экземпляр менеджера слежения за событиями
            wm = WatchManager()
            notifier = Notifier(wm, PC)

            print 'monitoring of %s started' % self.path

            added_flag = False
            # читать и обрабатывать события
            while True:
                try:
                    if not added_flag:
                        # на первой итерации добавить слежение за каталогом:
                        # обрабатываемые события определяются маской.
                        wm.add_watch(self.path, mask)
                        added_flag = True
                    notifier.process_events()
                    if notifier.check_events():
                        notifier.read_events()
                except KeyboardInterrupt:
                    # ...пока не будет нажата комбинация Ctrl-C
                    print 'stop monitoring...'
                    # прекратить слежение за событиями
                    notifier.stop()
                    break
                except Exception, err:
                    # продолжить слежение
                    print err

    def main():
        monitor = Controller()
        monitor.run()

    if __name__ == '__main__':
        main()
```

Если запустить этот сценарий, он начнет выполнять «требуемые» действия при помещении чего бы то ни было в каталог */tmp*. Этот пример должен дать вам некоторое представление о том, как фактически сделать что-нибудь полезное, например, добавить функцию обратного вызова для выполнения требуемых действий. Здесь же можно было использовать часть программного кода из главы «Данные», например, для автоматического поиска и удаления дубликатов или для архивирования файлов, если их имена соответствуют определяемому вами критерию в функции `fnmatch()`. В общем, это интересный и полезный модуль Python, который работает только в Linux.

OS X

OS X является довольно экзотической операционной системой, если не сказать больше. С одной стороны, она обладает, пожалуй, самым лучшим пользовательским интерфейсом Cocoa, а с другой, в версии Leopard, она является полностью POSIX-совместимой операционной системой UNIX. Системе OS X удалось добиться того, чего не удалось ни одному производителю операционных систем UNIX: она вывела UNIX на уровень массового потребления. Версия OS X Leopard включает в себя Python 2.5.1, Twisted и многие другие замечательные программные компоненты на языке Python.

Разработчики системы OS X следуют несколько странной парадигме, предлагая операционную систему и в серверном, и в обычном исполнении. Хотя, безусловно, компания Apple имеет на это полное право, возможно, ей стоит отказаться от такой архаичной идеи; мы же здесь не будем обсуждать плюсы и минусы концепции единой ОС по единой цене. Серверная версия операционной системы предлагает более полный комплект инструментов командной строки для администрирования, а также ряд компонентов, характерных для Apple, таких как возможность загрузки по сети, возможность работы с серверами каталогов LDAP, и многие другие особенности.

Взаимодействие с DSCL, утилитой службы каталогов

Название DSCL происходит от Directory Services Command Line (командная строка службы каталогов) и представляет собой удобный способ доступа к прикладному интерфейсу службы каталогов в OS X. DSCL позволяет читать, создавать и удалять записи, что язык Python позволяет делать естественным образом. В примере 8.10 демонстрируется взаимодействие с DSCL в оболочке IPython для чтения атрибутов службы Open Directory и их значений.

Обратите внимание, что в этом примере мы только читаем значения атрибутов, но точно так же, используя тот же прием, можно было бы организовать и их изменение.

Пример 8.10. Получение записи пользователя в интерактивной оболочке IPython с помощью DSCL

```
In [40]: import subprocess

In [41]: p = subprocess.Popen("dscl . read /Users/ngift",
shell=True,stdout=subprocess.PIPE)

In [42]: out = p.stdout.readlines()

In [43]: for line in out:
line.strip().split()

Out[46]: ['NFSHomeDirectory:', '/Users/ngift']
```

```
Out[46]: ['Password:', '********']
Out[46]: ['Picture:']
Out[46]: ['/Library/User', 'Pictures/Flowers/Sunflower.tif']
Out[46]: ['PrimaryGroupID:', '20']
Out[46]: ['RealName:', 'ngift']
Out[46]: ['RecordName:', 'ngift']
Out[46]: ['RecordType:', 'dsRecTypeStandard:Users']
Out[46]: ['UniqueID:', '501']
Out[46]: ['UserShell:', '/bin/zsh']
```

Это замечательно, что в Apple организовали централизованное управление локальными учетными записями и учетными записями LDAP/Active Directory с помощью команды `dscl`. Утилита `dscl` – это как глоток свежего воздуха по сравнению с другими средствами управления LDAP, даже если вынести использование Python за скобки. У нас недостаточно места, чтобы углубляться в подробности. Тем не менее, заметим, что с помощью языка Python и утилиты `dscl` можно очень легко организовать программное управление учетными записями как в локальной базе данных, так и в базе данных LDAP, такой как Open Directory, а предыдущий пример должен показать вам, с чего следует начинать.

Взаимодействие с прикладным интерфейсом OS X

Часто администратору OS X бывает необходимо знать, как организовать взаимодействие с фактическим интерфейсом пользователя. В OS X Leopard для языков Python и Ruby предоставляется доступ к механизму Scripting Bridge. За дополнительной информацией по этому механизму обращайтесь по адресу *http://developer.apple.com/documentation/Cocoa/Conceptual/RubyPythonCocoa/Introduction/Introduction.html*.

Как вариант, для доступа к OSA, или Open Scripting Architecture (открытая архитектура сценариев), можно использовать модуль `py-appscript` со страницей проекта по адресу *http://sourceforge.net/projects/appscript*.

Работать с модулем `py-appscript` – одно удовольствие, так как он дает возможность из языка Python взаимодействовать с очень богатой возможностями архитектурой OSA. Но прежде чем познакомится с ним поближе, мы сначала воспользуемся простым инструментом командной строки `osascript`, на примере которого продемонстрируем, как можно организовать взаимодействие с прикладным интерфейсом сценариев. В OS X Leopard теперь имеется возможность писать инструменты командной строки, работающие под управлением `osascript`, и выполнять их как обычные сценарии Bash или Python. Давайте напишем сценарий с именем `bofh.osa`, как показано ниже, и затем запустим его. Текст сценария приводится в примере 8.11.

Пример 8.11. Сценарий «Hello, Bastard Operator From Hell»

```
#!/usr/bin/osascript
say "Hello, Bastard Operator From Hell" using "Zarvox"
```

Если запустить этот сценарий из командной строки, механический голос поприветствует нас. Это немножко глупо, но это же Mac OS X; она вполне допускает такие вещи.

А теперь погрузимся в использование модуля `appscript` для доступа к тому же самому API из сценариев на языке Python, но сделаем это в интерактивном режиме, в оболочке IPython. Ниже представлена интерактивная версия примера, включенного в исходные тексты `appscript`, который выводит список всех запущенных процессов в алфавитном порядке:

```
In [4]: from appscript import app

In [5]: sysevents = app('System Events')

In [6]: processnames = sysevents.application_processes.name.get()

In [7]: processnames.sort(lambda x, y: cmp(x.lower(), y.lower()))

In [8]: print '\n'.join(processnames)
Activity Monitor
AirPort Base Station Agent
AppleSpell
Camino
DashboardClient
DashboardClient
Dock
Finder
Folder Actions Dispatcher
GrowlHelperApp
GrowlMenu
iCal
iTunesHelper
JavaApplicationStub
loginwindow
mdworker
PandoraBoy
Python
quicklookd
Safari
Spotlight
System Events
SystemUIServer
Terminal
TextEdit
TextMate
```

Если вам придется решать задачи автоматизации с применением приложений OS X, модуль `appscript` окажется для вас удачной находкой, так как с его помощью в языке Python можно реализовать такие действия, которые ранее были возможны только в языке Applescript. Ноа Гифт (Noah Gift) написал статью, в которой немного рассказывается

об этом: *http://www.macdevcenter.com/pub/a/mac/2007/05/08/using-python-and-applescript-to-get-the-most-out-of-your-mac.html*.

Кое-что системный администратор может выполнять с помощью Final Cut Pro, создавая пакеты операций, взаимодействующих, например, с Adobe After Effects. Кроме того, в OS X с помощью Applescript Studio можно быстро создать графический интерфейс и вызывать из него сценарий на языке Python командой `do shell script`. Мало кому известно, что оригинальная версия Carbon Copy Cloner была написана в Applescript Studio. Если у вас есть свободное время, вам стоит познакомиться с этой средой поближе.

Автоматическое восстановление системы

ASR – это еще один революционный, опередивший время инструмент командной строки, разработанный для OS X. Этот инструмент является ключевым компонентом очень популярной утилиты с именем Carbon Copy Cloner и служит для автоматизации многих ситуаций. Ноа (Noah) использовал утилиту `asr` в паре с Netboot для автоматического восстановления – фактически он ввел полную автоматизацию этого процесса в одном из мест, где он работал. Пользователю достаточно было просто перезагрузить свою машину и удерживать клавишу «N», чтобы перейти в режим загрузки по сети, и в результате либо наступал «конец игры», либо машина сама исправляла повреждения.

Пожалуйста, не рассказывайте об этом никому, потому что многие до сих пор думают, что он все еще работает там. Ниже, в примере 8.12 приводится упрощенная версия сценария для автоматического восстановления системы, который может быть запущен при загрузке по сети или со второго раздела жесткого диска. С точки зрения настроек, каталог */Users*, как и любой другой жизненно важный каталог, должен быть символической ссылкой, ведущей в другой раздел, или должен находиться в сети, что еще лучше. Смотрите пример 8.12.

Пример 8.12. Сценарий автоматического восстановления раздела жесткого диска в OS X, демонстрирующий ход выполнения с помощью виджета из библиотеки WXPython

```
#!/usr/bin/env pythonw
#автоматически восстанавливает раздел жесткого диска

import subprocess
import os
import sys
import time
from wx import PySimpleApp, ProgressDialog, PD_APP_MODAL, PD_ELAPSED_TIME

#команда пересоздания главного раздела с помощью утилиты asr
asr = '/usr/sbin/asr -source '

#переменные, содержащие различные пути
os_path = '/Volumes/main'
```

```
ipath = '/net/server/image.dmg '
dpath = '-target /Volumes/main -erase -noprompt -noverify &'
reimage_cmd = "%s%s%s" % (asr,ipath, dpath)

#Команды перезагрузки
reboot = 'reboot'
bless = '/usr/sbin/bless -folder /Volumes/main/System/Library/CoreServices -
setOF'

#часть использования wxpython
application = PySimpleApp()
dialog = ProgressDialog ('Progress', 'Attempting Rebuild of Main Partition',
                      maximum = 100, style = PD_APP_MODAL | PD_ELAPSED_TIME)

def boot2main():
    """Делает новый раздел загружаемым и выполняет перезагрузку"""
    subprocess.call(bless, shell=True)
    subprocess.call(reboot, shell=True)

def rebuild():
    """Пересоздает раздел"""
    try:
        time.sleep(5) #Дать диалогу время на запуск
        subprocess.call(reimage_cmd)
    except OSError:
        print "CMD: %s [ERROR: invalid path]" % reimage_cmd
        sys.exit(1)
    time.sleep(30)
    while True:
        if os.path.exists(os_path):
            x = 0
            wxSleep(1)
            dialog.Update (x + 1,
                      "Rebuild is complete...\n rebooting to main partition\n
                       ...in 5 seconds..")
            wxSleep(5)
            print "repaired volume.." + os_path
            boot2main() #вызывает функции reboot/bless
            break
        else:
            x = 0
            wxSleep(1)
            dialog.Update ( x + 1, 'Reimaging.... ')

def main():
    if os.path.exists(os_path):
        rebuild()
    else:
        print "Could not find valid path...FAILED.."
        sys.exit(1)

if __name__ == "__main__":
    main()
```

Этот сценарий пытается пересоздать раздел и выводит средствами библиотеки WXPython индикатор хода выполнения. Если путь указан корректно и не обнаружено ошибок, выполняется пересоздание раздела жесткого диска с помощью команды `asr`, в процессе выполнения которой выводится индикатор, показывающий ход выполнения операции, затем новый раздел назначается загружаемым с помощью команды `bless`, после чего выполняется перезагрузка машины.

Этот сценарий легко можно превратить в основу системы управления и распространения дистрибутива системы уровня предприятия, поскольку достаточно легко организовать установку различных образов, основываясь на данных об аппаратной комплектации или даже считывая «старую» метку жесткого диска. После этого можно, например, организовать программную установку пакетов программного обеспечения с помощью системы управления пакетами в OS X или с помощью свободно распространяемого инструмента radmind. Hoa (Noah) реализовал один интересный сценарий, в котором сначала в автоматическом режиме развертывал базовую систему OS X, а затем завершал установку остальных пакетов с помощью radmind.

Если вы всерьез собираетесь заниматься администрированием систем OS X, то вам определенно стоило бы поближе познакомиться с radmind. Radmind – это своего рода система автоматического обновления, которая обнаруживает изменения в файловой системе и обеспечивает возможность восстановления машин на основе этих изменений. Дополнительную информацию о radmind вы найдете на странице *http://rsug.itd.umich.edu/software/radmind/*. Несмотря на то, что программа radmind написана не на языке Python, ее легко можно было бы переписать на этом языке.

Управление файлами Plist из сценариев на языке Python

В главе 3 мы выполняли анализ потока информации в формате XML, генерируемого утилитой `system_profiler`, используя для этого библиотеку ElementTree. Но в OS X в Python встроена поддержка библиотеки `plistlib`, которая позволяет анализировать и создавать файлы Plist. Сам модуль тоже называется `plistlib`. У нас нет возможности продемонстрировать этот модуль на примерах, но вам стоит познакомиться с ним поближе самостоятельно.

Администрирование систем Red Hat Linux

В Red Hat язык Python используется очень широко – и в компании, и в операционной системе. Некоторые из наиболее интересных новых способов использования Python родились в группе Emerging Technologies: *http://et.redhat.com/page/Main_Page*. Ниже приводится список некоторых проектов, использующих язык Python:

- Libvert – API виртуализации менеджера виртуальных машин

- VirtInst – приложение управления виртуальными машинами на базе библиотеки libvirt[1], написанное на языке Python + PyGTK
- Библиотека Python, упрощающая инициализацию гостевых виртуальных машин на основе libvirt
- Cobbler – продукт, позволяющий создавать полностью автоматизированные серверы загрузки для нужд PXE и виртуализации
- Virt-Factory: сетевая среда управления виртуальными приложениями
- FUNC (Fedora Unified Network Controller)

Администрирование Ubuntu

Можно сказать, что из всех основных дистрибутивов Linux Ubuntu является одним из самых влюбленных в Python. Отчасти потому, что Марк Шаттлворт (Mark Shuttleworth), создатель дистрибутива, долгое время – с начала 90 годов – работал с языком Python. Одним из замечательных источников пакетов на языке Python для Ubuntu является Launchpad: *http://launchpad.net*.

Администрирование систем Solaris

С конца 90-х до начала 2000-х годов операционная система Solaris занимала практически непоколебимое место в мире UNIX. В начале 2000-х годов Linux подобно метеориту врезался в Solaris, в связи с чем компании Sun пришлось испытать вполне реальные неприятности. Однако с недавнего времени все больше системных администраторов, разработчиков и предпринимателей снова начинают говорить о Solaris.

Из наиболее интересных нововведений, которые предполагает внести Sun, можно назвать 6-месячный цикл выпуска новых версий системы, так же, как и в Ubuntu, с 18-месячным периодом технической поддержки; отказ от создания объемного дистрибутива на DVD-диске в пользу единственного CD, как в Ubuntu. Наконец, были заимствованы некоторые идеи из Red Hat и Fedora по созданию версии Solaris, разрабатываемой сообществом. Загрузить или заказать загрузочный CD можно по адресу: *http://www.opensolaris.com*.

Что все это означает для системного администратора, использующего язык Python? Интерес к Sun быстро растет, и у нее имеется большое количество весьма интересных технологий, начиная от ZFS и заканчивая контейнерами и LDOM, которые в некотором смысле можно сравнить с виртуальными машинами VMware. Имеется даже связь с этой книгой. Интерпретатор Python прекрасно работает в операционной

[1] libvert и libvirt – это разные библиотеки! – *Прим. перев.*

системе Solaris и даже широко используется в разработке системы управления пакетами для нее.

Виртуализация

14 августа 2007 года состоялось первое открытое размещение акций компании VMware, которое принесло ей миллионы долларов и укрепило позиции виртуализации как крупного направления в развитии информационных технологий. Предсказание будущего всегда было рискованным делом, однако все чаще в крупных компаниях слышны слова «операционная система центра обработки данных», и все, от Microsoft до Red Hat и Oracle, стремятся не опоздать сесть в поезд виртуализации. Можно смело сказать, что со временем виртуализация полностью изменит центры обработки данных и работу системных администраторов. Виртуализация – это элементарный пример действия слишком частого использования фразы «прорывная технология».

Виртуализация – это обоюдоострое оружие для системных администраторов, так как, с одной стороны, позволяет легко тестировать системы и приложения, но, с другой стороны, чрезвычайно увеличивает сложность администрирования. Теперь на одной машине одновременно может работать сразу несколько операционных систем, здесь могут находиться приложения для малого бизнеса или крупная часть вычислительного центра. За эффективность приходится платить, а обеспечение эффективности – прямая обязанность среднего системного администратора.

Возможно, прямо сейчас, читая эти строки, вы могли бы подумать: какое отношение все это имеет к языку Python? Самое непосредственное. В компании Racemi, где недавно работал Hoa (Noah), на языке Python было написано полноценное приложение управления центром обработки данных, которое имеет дело с виртуализацией. Python может и действительно очень тесно взаимодействует с механизмами виртуализации, начиная от управления виртуальными машинами и заканчивая перемещением систем с физических машин на виртуальные, используя для этого Python API. В этом виртуализованном мире Python чувствует себя как дома и можно смело утверждать, что он будет играть далеко не последнюю роль в будущей операционной системе центра обработки данных.

VMware

Как уже говорилось выше, компания VMware является лидером по разработке технологий виртуализации. Наличие полного программного контроля над виртуальной машиной – это своего рода Чаша Грааля. К счастью, существует несколько API на выбор: Perl, XML-RPC, Python и C. К моменту написания этих строк некоторые реализации Python имели определенные ограничения, но такое положение дел могло

измениться. Похоже, что в VMware выбрали новое направление – XML-RPC API.

Компания VMware выпускает несколько различных продуктов с различными API. Из продуктов, с которыми вам может потребоваться взаимодействовать, можно назвать VMware Site Recovery Manager, VMware ESX Server, VMware Server и VMware Fusion.

У нас недостаточно места, чтобы охватить принципы взаимодействия с этими технологиями, поскольку эта тема выходит далеко за рамки данной книги, но они стоят того, чтобы следить за их развитием и за тем, какую роль будет играть Python.

Облачная обработка данных

Только-только утихла шумиха вокруг виртуализации, как вдруг возник шум об «облачной» обработке данных (cloud computing). Термин «облачная обработка данных» обозначает технологию выделения вычислительных ресурсов по требованию, в зависимости от величины рабочей нагрузки. В сфере развития технологии «облачной» обработки данных присутствуют два крупных игрока – Amazon и Google. Буквально за несколько недель до передачи этой книги издателю компания Google взорвала настоящую бомбу. Компания предложила интереснейшую «фишку», которая пока поддерживается только языком Python. Поскольку эта книга посвящена языку Python, мы думаем, что такое ограничение не слишком огорчит вас. В некотором смысле такое предпочтение, отданное языку Python, напоминает нам рекламу American Express.

В этом разделе мы пройдемся по некоторым имеющимся API, с которыми вам придется столкнуться при работе с Amazon и с Google App Engine. В заключение мы поговорим о том, как это может касаться системных администраторов.

Веб-службы Amazon на основе Boto

Отличную возможность для работы с инфраструктурой «облачной» обработки данных Amazon предоставляет интерфейс Boto. Посредством Boto обеспечивается доступ к таким службам, как Simple Storage Service, Simple Queue Service, Elastic Compute Cloud, Mechanical Turk, SimpleDB. Это совершенно новый и очень мощный API, поэтому мы рекомендуем заглянуть на домашнюю страницу проекта *http://code.google.com/p/boto/*. Здесь вы сможете почерпнуть самую свежую информацию, что лучше, чем приведение нами сведений, доступных на данный момент.

Ниже приводится короткий пример взаимодействия со службой SimpleDB.

Соединение со службой:

```
In [1]: import boto
In [2]: sdb = boto.connect_sdb()
```

Создание нового домена:

```
In [3]: domain = sdb.create_domain('my_domain')
```

Добавление нового элемента:

```
In [4]: item = domain.new_item('item')
```

Примерно так выглядит API в настоящее время, но, чтобы получить полное представление, вам необходимо взглянуть на примеры в репозитарии svn: *http://code.google.com/p/boto/source/browse*. Заметим, что изучение примеров – это один из лучших способов понять, как работает новая библиотека.

Google App Engine

Служба Google App Engine выпущена в состоянии бета-версии и со дня объявления была широко разрекламирована. Она позволяет свободно запускать приложения в инфраструктуре Google. Приложения App Engine имеют API пока только для Python, но со временем такое положение дел может измениться. Одна из интереснейших особенностей App Engine заключается в том, что она интегрирована с другими службами Google.

Все более возможным становится перемещение большей части из того, что находилось у вас в центре обработки данных, в другие центры, поэтому это все более затрагивает системных администраторов. Умение взаимодействовать со службой Google App Engine может оказаться ка-

ПОРТРЕТ ЗНАМЕНИТОСТИ: КОМАНДА РАЗРАБОТЧИКОВ GOOGLE APP ENGINE

Кевин Гиббс (Kevin Gibbs)

Кевин Гиббс – технический лидер проекта Google App Engine. Кевин присоединился к Google в 2004 году. До работы над Google App Engine в течение ряда лет работал в группе разработки инфраструктуры систем, где занимался системами управления кластерами, которые составляют основу продуктов и служб компании Google. Кроме того, Кевин является автором Google Suggest, программного продукта, обеспечивающего вывод интерактивных подсказок в процессе ввода с клавиатуры. До присоединения к Google Кевин работал в группе передовых интернет-технологий компании IBM, где занимался созданием инструментов разработчика.

чественно новым навыком для системного администратора, поэтому есть смысл заняться ее исследованием.

Мы побеседовали с некоторыми специалистами из команды разработчиков App Engine и спросили их о том, что в первую очередь может пригодиться системным администраторам. Они выделили следующие задачи:

1. Выгрузка больших объемов данных: *http://code.google.com/appengine/articles/bulkload.html*.

   Системным администраторам часто приходится перемещать огромные объемы данных, и этот инструмент позволит решать эти проблемы в контексте приложений из Google App Engine.

2. Регистрация событий: *http://code.google.com/appengine/articles/logging.html*.

3. Интерфейс к электронной почте: функция `send_mail_to_admin()`: *http://code.google.com/appengine/docs/mail/functions.html*.

   С точки зрения системного администратора владение этим интерфейсом может оказаться полезным для организации мониторинга. В случае появления важных исключений или выполнения операций вы могли бы автоматически отправлять администраторам приложений сообщения по электронной почте.

4. Выполнение периодических задач с помощью планировщика заданий cron.

   Это не является непосредственной частью Google App Engine, но вы могли бы использовать планировщик заданий cron на своих серверах для передачи запросов своим приложениям через определенные интервалы времени. Например, можно было бы оформить задание для планировщика, согласно которому каждый час будет посылаться запрос по адресу *http://yourapp.com/emailsummary*, в результате которого системному администратору будет высылаться сообщение электронной почты с описанием важных событий, произошедших в течение последнего часа.

5. Управление версиями: *http://code.google.com/appengine/docs/configuringa napp.html#Required_Elements*.

   Одно из обязательных полей, заполняемых для вашего приложения, – идентификатор версии. Каждый раз, когда выгружается приложение с тем же идентификатором версии, оно замещается новым программным кодом. Изменяя идентификатор версии, вы получаете возможность иметь несколько версий приложения и с помощью консоли администратора выбирать, какая из версий должна быть включена в работу.

Создание примера приложения для Google App Engine

Прежде чем приступить к созданию приложения для Google App Engine, вам потребуется загрузить пакет SDK для Google App Engine:

http://code.google.com/appengine/downloads.html. Вы также можете ознакомиться с замечательным учебным руководством по Google App Engine: *http://code.google.com/appengine/docs/gettingstarted/*.

В этом разделе мы предлагаем обратное учебное руководство для Google App Engine, так как замечательное учебное руководство уже существует. Если вы перейдете по адресу *http://greedycoin.appspot.com/*, то сможете опробовать работающую версию приложения, которое описывается ниже, а также познакомиться с последней версией исходных текстов. Приложение принимает сумму, введенную пользователем, сохраняет ее в базе данных и затем возвращает сумму в виде списка монет определенного достоинства. Приложением также поддерживается возможность регистрации посредством API аутентификации и возможность получения информации о последних запросах. Исходный текст приложения приводится в примере 8.13.

Пример 8.13. Веб-приложение Greedy Coin

```
#!/usr/bin/env python2.5
#Noah Gift

import decimal
import wsgiref.handlers
import os

from google.appengine.api import users
from google.appengine.ext import webapp
from google.appengine.ext import db
from google.appengine.ext.webapp import template

class ChangeModel(db.Model):
    user = db.UserProperty()
    input = db.IntegerProperty()
    date = db.DateTimeProperty(auto_now_add=True)

class MainPage(webapp.RequestHandler):
    """Главная страница"""

    def get(self):
        user = users.get_current_user()

        if users.get_current_user():
            url = users.create_logout_url(self.request.uri)
            url_linktext = 'Logout'
        else:
            url = users.create_login_url(self.request.uri)
            url_linktext = 'Login'

        template_values = {
        'url': url,
        'url_linktext': url_linktext,
        }
        path = os.path.join(os.path.dirname(__file__), 'index.html')
        self.response.out.write(template.render(path, template_values))
```

```
class Recent(webapp.RequestHandler):
    """Получение информации о 10 последних запросах"""

    def get(self):
        #коллекция
        collection = []
        #получить 10 последних записей из хранилища данных
        query = ChangeModel.all().order('-date')
        records = query.fetch(limit=10)

        #отформатировать дробные значения
        for change in records:
            collection.append(decimal.Decimal(change.input)/100)

        template_values = {
        'inputs': collection,
        'records': records,
        }

        path = os.path.join(os.path.dirname(__file__), 'query.html')
        self.response.out.write(template.render(path,template_values))

class Result(webapp.RequestHandler):
    """Возвращает страницу с результатами"""

    def __init__(self):
        self.coins = [1,5,10,25]
        self.coin_lookup = {25: "quarters", 10: "dimes", 5: "nickels",
                            1: "pennies"}

    def get(self):
        #Просто получить последнее число
        collection = {}

        #выбрать последний ввод из хранилища данных
        change = db.GqlQuery(
            "SELECT * FROM ChangeModel ORDER BY date DESC LIMIT 1")
        for c in change:
            change_input = c.input

        #логика размена суммы монетами
        coin = self.coins.pop()
        num, rem = divmod(change_input, coin)
        if num:
            collection[self.coin_lookup[coin]] = num
        while rem > 0:
            coin = self.coins.pop()
            num, rem = divmod(rem, coin)
            if num:
                collection[self.coin_lookup[coin]] = num

        template_values = {
        'collection': collection,
        'input': decimal.Decimal(change_input)/100,
        }
```

```
        #шаблон отображения
        path = os.path.join(os.path.dirname(__file__), 'result.html')
        self.response.out.write(template.render(path, template_values))

class Change(webapp.RequestHandler):

    def post(self):
        """метод вывода результатов"""
        model = ChangeModel()
        try:
            change_input = decimal.Decimal(self.request.get('content'))
            model.input = int(change_input*100)
            model.put()
            self.redirect('/result')
        except decimal.InvalidOperation:
            path = os.path.join(os.path.dirname(__file__),
                                'submit_error.html')
            self.response.out.write(template.render(path,None))

def main():
    application = webapp.WSGIApplication([('/', MainPage),
                                          ('/submit_form', Change),
                                          ('/result', Result),
                                          ('/recent', Recent)],
                                          debug=True)
    wsgiref.handlers.CGIHandler().run(application)

if __name__ == "__main__":
    main()
```

Так как это обратное учебное руководство, начнем с рассмотрения версии приложения, работающей по адресу *http://greedycoin.appspot.com/*, или с вашей версии по адресу *http://localhost:8080/*. На главной странице приложения на фоне цвета тыквы находятся две прямоугольные области: область слева представляет собой форму, где вы можете ввести денежную сумму, а область справа содержит элементы навигации. Эти приятные (или уродливые) цвета и схема размещения являются комбинацией задействованного механизма шаблонов Django и CSS. Шаблоны Django можно найти в главном каталоге, а используемые CSS – в таблицах стилей. Механизм оформления не имеет никакого отношения к Google App Engine, поэтому за дополнительной информацией о механизме шаблонов Django мы просто отсылаем вас к руководству: *http://www.djangoproject.com/documentation/templates/*.

Теперь, когда мы познакомились с внешним видом приложения, давайте перейдем к изучению некоторых особенностей Google App Engine. Обратите внимание на ссылку «Login» в правой области: она обеспечивает возможность использовать прикладной интерфейс механизма аутентификации. Ниже показано, как это реализовано в программном коде:

```
class MainPage(webapp.RequestHandler):
    """Главная страница"""

    def get(self):
        user = users.get_current_user()

        if users.get_current_user():
            url = users.create_logout_url(self.request.uri)
            url_linktext = 'Logout'
        else:
            url = users.create_login_url(self.request.uri)
            url_linktext = 'Login'

        template_values = {
        'url': url,
        'url_linktext': url_linktext,
        }
        path = os.path.join(os.path.dirname(__file__), 'index.html')
        self.response.out.write(template.render(path, template_values))
```

Здесь представлен класс, наследующий свойства и методы класса `webapp.RequestHandler`, и если вы определите метод `get()`, вы сможете проверять, зарегистрировался ли пользователь. Если вы посмотрите на несколько последних строк, то увидите, что информация о пользователе помещается в шаблон, который затем используется механизмом шаблонов Django для отображения страницы *index.html*. Это просто замечательно, что таким тривиальным способом можно задействовать мощную базу учетных записей Google для обеспечения возможности авторизации на страницах. В предыдущем фрагменте это взаимодействие достигается всего двумя строчками:

```
user = users.get_current_user()

if users.get_current_user():
```

Здесь мы могли бы предложить вам поэкспериментировать с этим фрагментом и попытаться изменить его так, чтобы приложение было доступно только для зарегистрировавшихся пользователей. Для этого вам даже не требуется понимать, как работает весь механизм, – достаточно будет использовать уже имеющиеся условные инструкции.

Теперь, когда мы получили некоторое представление об аутентификации, перейдем к вещам более сложным. Прикладной интерфейс к хранилищу данных позволяет сохранять данные и затем извлекать их в любой части приложения. Для этого необходимо импортировать модуль `db`, как показано в предыдущем примере, и определить модель:

```
class ChangeModel(db.Model):
    user = db.UserProperty()
    input = db.IntegerProperty()
    date = db.DateTimeProperty(auto_now_add=True)
```

С помощью этого простого класса мы можем создавать и использовать хранимые данные. Ниже приводится класс, в котором используется Python API для получения данных из хранилища и отображения 10 последних результатов:

```
class Recent(webapp.RequestHandler):
    """Получение информации о 10 последних запросах"""

    def get(self):

        #коллекция
        collection = []
        #получить 10 последних записей из хранилища данных
        query = ChangeModel.all().order('-date')
        records = query.fetch(limit=10)

        #отформатировать дробные значения
        for change in records:
            collection.append(decimal.Decimal(change.input)/100)

        template_values = {
        'inputs': collection,
        'records': records,
        }

        path = os.path.join(os.path.dirname(__file__), 'query.html')
        self.response.out.write(template.render(path,template_values))
```

Самые важные строки здесь:

```
query = ChangeModel.all().order('-date')
records = query.fetch(limit=10)
```

Они выбирают результаты из хранилища данных и затем «извлекают» (fetch) 10 последних записей в запросе. Здесь можно остановиться и поэкспериментировать с этим фрагментов, попытавшись получить большее число записей или отсортировать их в другом порядке. Это должно дать вам прочувствовать взаимодействие с приложением.

Наконец, если внимательно посмотреть на фрагмент ниже, можно обнаружить, что каждому URL в списке соответствует свой класс, который определен в нашем файле *change.py*. Здесь мы могли бы порекомендовать вам поэкспериментировать с URL, изменяя соответствия между ними и частями приложения – это должно дать представление о том, как они задействуются.

```
def main():
    application = webapp.WSGIApplication([('/', MainPage),
        ('/submit_form', Change),
        ('/result', Result),
        ('/recent', Recent)],
        debug=True)
    wsgiref.handlers.CGIHandler().run(application)
```

На этом мы заканчиваем наше обратное учебное руководство по Google App Engine, которое должно было дать вам некоторое представление о том, как можно было бы реализовать собственный инструмент для нужд системного администрирования. Если вам интересно будет ознакомиться с примерами других приложений, можете также познакомиться с исходными текстами приложения Google App Engine, написанного самим Гвидо ван Россумом (Guido van Rossum): *http://code.google.com/p/rietveld/source/browse*.

Использование Zenoss для управления серверами Windows из Linux

Если вы имеете несчастье заниматься администрированием одного или нескольких серверов, работающих под управлением Windows, ваша работа может стать немного менее неприятной. В этом нам может помочь такой удивительный инструмент, как Zenoss. Мы говорили о Zenoss в главе 7 «SNMP». Помимо того, что он является инструментом SNMP, он к тому же обеспечивает возможность взаимодействовать с серверами Windows через WMI (Windows Management Interface – интерфейс управления Windows) из Linux! Мы можем только посмеиваться и размышлять о практических применениях, полагаясь на эти технологии. Из разговоров со специалистами проекта Zenoss мы выяснили, что они предлагают передавать сообщения WMI серверу Samba (или CIFS) на машине, работающей под управлением Linux, и посылать их серверу Windows. И, пожалуй, самое интересное (по крайней мере, для читателей этой книги) заключается в том, что имеется возможность организовать взаимодействие с соединением WMI из сценариев на языке Python.

Обсуждение синтаксиса и особенностей WMI выходит далеко за рамки этой книги.

Существующая документация к Zenoss прекрасно освещает принципы взаимодействия с WMI из Linux с помощью языка Python. Тем не менее, примеры, которые мы собираемся представить вашему вниманию, должны послужить хорошей основой для вашего дальнейшего усовершенствования. Для начала рассмотрим применение инструмента `wmic` (не имеющего отношения к языку Python) для обеспечения взаимодействия с сервером Windows через WMI из операционной системы Linux. `wmic` – это простая утилита командной строки, которая принимает в качестве аргументов командной строки имя пользователя, пароль, адрес сервера и запрос WMI. Она выполняет подключение к указанному серверу с заданными параметрами аутентификации, передает запрос и отображает результаты на устройстве стандартного

вывода. Синтаксис использования этой утилиты выглядит следующим образом:

```
wmic -U username%password //SERVER_IP_ADDRESS_OR_HOSTNAME "some WMI query"
```

В следующем примере выполняется соединение с сервером, имеющим IP-адрес 192.168.1.3, с именем пользователя Administrator, и производится запрос на получение записей из журнала событий:

```
wmic -U Administrator%password //192.168.1.3 "SELECT * FROM Win32_NTLogEvent"
```

А ниже приводится часть результатов, полученных в ходе выполнения этой команды:

```
CLASS: Win32_NTLogEvent
Category|CategoryString|ComputerName|Data|EventCode|EventIdentifier|
    EventType|InsertionStrings|Logfile|Message|RecordNumber|SourceName|
    TimeGenerated|TimeWritten|Type|User
...
|3|DCOM|20080320034341.000000+000|20080320034341.000000+000|Information|(null)
0|(null)|MACHINENAME|NULL|6005|2147489653|3|(,,,,14,0,0 )|System|The Event log
service was started.
|2|EventLog|20080320034341.000000+000|20080320034341.000000+000|Information|
(null)0|(null)|MACHINENAME|NULL|6009|2147489657|3|(5.02.,3790,Service Pack
2,Uniprocessor Free)|System|Microsoft (R) Windows (R) 5.02. 3790 Service Pack 2
Uniprocessor Free.
|1|EventLog|20080320034341.000000+000|20080320034341.000000+000|Information|
(null)
```

Чтобы выполнить аналогичный запрос из сценария на языке Python, сначала необходимо настроить окружение. Для примеров, следующих ниже, мы использовали комплекс Zenoss 2.1.3 VMware. В этом комплексе часть программного кода Zenoss располагается в домашнем каталоге пользователя zenoss. Самое сложное заключается в том, чтобы добавить путь к каталогу, где находится модуль wmiclient.py, в переменную окружения PYTHONPATH. Мы добавили путь к каталогу в начало уже существующей переменной PYTHONPATH, как показано ниже:

```
export PYTHONPATH=~/Products/ZenWin:$PYTHONPATH
```

Обеспечив возможность доступа к необходимым библиотекам, можно попробовать запустить сценарий, исходный текст которого приводится ниже:

```
#!/usr/bin/env python

from wmiclient import WMI

if __name__ == '__main__':
    w = WMI('winserver', '192.168.1.3', 'Administrator', passwd='foo')
    w.connect()
    q = w.query('SELECT * FROM Win32_NTLogEvent')
    for l in q:
```

```
        print "l.timewritten::", l.timewritten
        print "l.message::", l.message
```

Вместо того чтобы выводить значения всех полей, как это сделано в примере с применением `wmic`, этот сценарий выводит только время и текст сообщения из журнала. Данный сценарий соединяется с сервером `192.168.1.3` с привилегиями пользователя `Administrator` и с паролем `foo`. Затем он выполняет запрос WMI `'SELECT * FROM Win32_NTLogEvent'`. После этого производится обход полученных результатов и вывод времени и текста сообщения для каждой записи. Трудно придумать что-либо более простое, чем этот пример.

Ниже приводится часть вывода, полученного от этого сценария:

```
l.timewritten:: 20080320034359.000000+000
l.message:: While validating that \Device\Serial1 was really a serial port,
a fifo was detected. The fifo will be used.

l.timewritten:: 20080320034359.000000+000
l.message:: While validating that \Device\Serial0 was really a serial port,
a fifo was detected. The fifo will be used.

l.timewritten:: 20080320034341.000000+000
l.message:: The COM sub system is suppressing duplicate event log entries for
a duration of 86400 seconds. The suppression timeout can be controlled by
a REG_DWORD value named SuppressDuplicateDuration under the following registry
key: HKLM\Software\Microsoft\Ole\EventLog.

l.timewritten:: 20080320034341.000000+000
l.message:: The Event log service was started.

l.timewritten:: 20080320034341.000000+000
l.message:: Microsoft (R) Windows (R) 5.02. 3790 Service Pack 2 Uniprocessor
Free.
```

Но как мы узнали, что необходимо использовать атрибуты `timewritten` и `message`? Чтобы найти эту информацию, потребовалось приложить совсем немного усилий. Ниже приводится сценарий, который помогает отыскивать необходимые атрибуты:

```
#!/usr/bin/env python

from wmiclient import WMI

if __name__ == '__main__':
    w = WMI('winserver', '192.168.1.3', 'Administrator', passwd='foo')
    w.connect()
    q = w.query('SELECT * FROM Win32_NTLogEvent')
    for l in q:
        print "result set fields::->", l.Properties_.set.keys()
        break
```

Вы могли бы заметить, что этот сценарий очень похож на предыдущий сценарий WMI. Между этими сценариями имеются два отличия – данный сценарий вместо вывода значений времени и текста сообщения

выводит результат метода `l.Properties_.set.keys()` и прерывает цикл после вывода первого результата. Объект `set`, метод `keys()` которого мы вызываем, в действительности является словарем. (Что сразу приобретает определенный смысл, потому что `keys()` является методом словаря.) Каждая запись в результатах, полученных по запросу WMI, должна иметь ряд атрибутов, имена которых соответствуют ключам этого словаря. А теперь приведем результаты работы сценария, который мы только что обсудили:

```
result set fields::-> ['category', 'computername', 'categorystring',
'eventidentifier', 'timewritten', 'recordnumber', 'eventtype', 'eventcode',
'timegenerated', 'sourcename', 'insertionstrings', 'user', 'type',
'message',
'logfile', 'data']
```

Как видите, оба атрибута, `timewritten` и `message`, использованные нами в первом сценарии WMI, присутствуют в списке ключей.

Мы не считаем себя большими поклонниками работы с операционной системой Windows, но тем не менее, мы понимаем, что иногда для выполнения работы приходится использовать предопределенные технологии. Этот инструмент от Zenoss поможет сделать решение такого рода задач менее неприятным делом. К тому же этот инструмент обладает такими широкими возможностями, что позволяет выполнять запросы WMI из Linux. Если вам приходится работать с операционной системой Windows, то Zenoss с успехом может занять видное место в вашем инструментарии.

9

Управление пакетами

Введение

Управление пакетами является одной из наиболее важных составляющих успешного проектирования программного обеспечения. Управление пакетами в разработке аналогично организации доставки в компаниях, занимающихся электронной торговлей, таких как Amazon. Если бы не было компаний, выполняющих доставку, то и существование самой компании Amazon было бы невозможным. Точно так же, если не будет простой, устойчивой и функциональной системы управления пакетами для операционной системы или языка, то и разработка программного обеспечения будет сталкиваться с определенными ограничениями.

Когда упоминается «управление пакетами», возможно, первое, о чем вы вспоминаете, это о пакетах *.rpm* и утилите `yum` или о пакетах *.deb* и утилите `apt`, или о каких-то других комплексах управления пакетами уровня операционной системы. Мы рассмотрим все это в данной главе, но основное наше внимание будет уделено созданию и управлению пакетами модулей на языке Python и среде окружения языка Python. При использовании Python всегда имелась возможность обеспечить доступ к программному коду на языке Python для всей системы. Кроме того, недавно появилось несколько проектов, которые еще больше улучшили гибкость, удобство и простоту создания пакетов с модулями на языке Python, управления ими и распространения.

Среди этих проектов можно назвать setuptools, Buildout и virtualenv. Часто эти проекты используют в процессе разработки и для управления средой разработки. Но по большей части они предназначены для обеспечения развертывания программного кода на языке Python платформенно-независимым способом. (Обратите внимание на оговорку «по большей части».)

Другой способ развертывания связан с созданием системно-зависимых пакетов и передачей их на компьютеры конечных пользователей. В некоторых случаях это два совершенно независимых подхода, хотя в них имеется и что-то общее. В этой главе мы будем рассматривать свободно распространяемый инструмент EPM, который способен создавать пакеты для платформ AIX, Debian/Ubuntu, FreeBSD, HP-UX, IRIX, Mac OS X, NetBSD, OpenBSD, Red Hat, Slackware, Solaris и Tru64 Unix.

Знание принципов управления пакетами необходимо не только разработчикам программного обеспечения. Это совершенно необходимо и системным администраторам. На практике нередко системный администратор является тем человеком, на которого возлагаются задачи управления пакетами. Понимание новейших приемов управления пакетами для языка Python и для различных операционных систем является одним из способов повысить свою ценность как специалиста. Хотелось бы надеяться, что данная глава поможет вам в этом. Кроме того, ценную информацию по темам, которые мы затронем здесь, можно найти по адресу *http://wiki.python.org/moin/buildout/pycon2008_tutorial*.

Setuptools и пакеты Python Eggs

Согласно официальной документации «setuptools – это набор расширений к distutils языка Python (на большинстве платформ – для Python 2.3.5, однако для 64-битовых платформ требуется версия не ниже Python 2.4), которые упрощают сборку и распространение пакетов, особенно когда они имеют зависимости от других пакетов».

До появления setuptools комплект distutils был основным средством создания установочных пакетов с модулями на языке Python. setuptools – это библиотека, которая расширяет возможности distutils. Название «eggs» относится к окончательному комплекту пакетов и модулей на языке Python, напоминая файлы *.rpm* или *.deb*. Как правило, они распространяются в формате архива ZIP и устанавливаются либо в сжатом виде, либо распаковываются, чтобы иметь возможность перемещаться по содержимому пакета. Создание пакетов «eggs» – это особенность библиотеки setuptools, которая работает с `easy_install`. Согласно официальной документации «Easy Install – это модуль на языке Python (`easy_install`), связанный с библиотекой setuptools, которая позволяет автоматически загружать, собирать, устанавливать и управлять пакетами языка Python». Несмотря на то, что это модуль, чаще его воспринимают и используют как инструмент командной строки. В этом разделе мы расскажем о `setuptools`, `easy_install` и `eggs` и разъясним, для чего каждый из этих инструментов используется.

В этой главе мы выделим наиболее полезные на наш взгляд особенности `setuptools` и `easy_install`. Чтобы получить полный комплект документации к ним, обращайтесь по адресам *http://peak.telecommunity.com/*

DevCenter/setuptools и *http://peak.telecommunity.com/DevCenter/EasyInstall*, соответственно.

Сложные инструменты, способные делать удивительные вещи, часто бывает сложно понять. Инструмент `setuptools` сложно понять отчасти потому, что он делает именно удивительные вещи. С помощью этого раздела, который можно рассматривать как краткое руководство, и с последующим изучением руководств вы получите возможность научиться использовать setuptools, easy_install и пакеты Python как пользователь и как разработчик.

Использование easy_install

Основные принципы использования `easy_install` понять очень легко. Большинство читателей этой книги наверняка использовали rpm, yum, apt-get, fink или подобные им инструменты управления пакетами. Фраза «Easy Install» (простая установка) часто означает использование инструмента командной строки с именем `easy_install` для выполнения задач, похожих на выполняемые утилитой `yum` в системах на базе Red Hat или `apt-get` в системах на базе Debian, – для пакетов Python.

Инструмент `easy_install` можно установить с помощью запуска сценария «начальной установки» с именем `ez_setup.py` для версии Python, с которой будет работать `easy_install`. Сценарий `ez_setup.py` загрузит последнюю версию setuptools и автоматически установит `easy_install` как сценарий в местоположение по умолчанию, которое в UNIX-подобных системах обычно соответствует каталогу, где находится исполняемый файл интерпретатора `python`. Давайте посмотрим, насколько это «просто» в действительности. Взгляните на пример 9.1.

Пример 9.1. Загрузка и установка easy_install

```
$ curl http://peak.telecommunity.com/dist/ez_setup.py
> ez_setup.py
% Total    % Received % Xferd  Average Speed   Time    Time    Time    Current
Dload  Upload   Total   Spent    Left  Speed
100  9419  100  9419    0     0    606     0   0:00:15 0:00:15 --:--:--  83353
$ ls
ez_setup.py
$ sudo python2.5 ez_setup.py
Password:
Searching for setuptools
Reading http://pypi.python.org/simple/setuptools/
Best match: setuptools 0.6c8
Processing setuptools-0.6c8-py2.5.egg
setuptools 0.6c8 is already the active version in easy-install.pth
Installing easy_install script to /usr/local/bin
Installing easy_install-2.5 script to /usr/local/bin
```

```
Using /Library/Python/2.5/site-packages/setuptools-0.6c8-py2.5.egg
Processing dependencies for setuptools
Finished processing dependencies for setuptools
$
```

В этом случае сценарий `easy_install` был помещен в каталог */usr/local/bin* под двумя различными именами.

```
$ ls -l /usr/local/bin/easy_install*
-rwxr-xr-x   1 root  wheel  364 Mar  9 18:14 /usr/local/bin/easy_install
-rwxr-xr-x   1 root  wheel  372 Mar  9 18:14 /usr/local/bin/easy_install-2.5
```

В соответствии с соглашениями, продолжительное время существующими в языке Python, при установке выполняемых файлов программ один файл устанавливается под именем, содержащим номер версии Python, и один – без номера версии. Это означает, что по умолчанию будет использоваться файл, имя которого не содержит номер версии, пока пользователь явно не укажет имя файла с номером версии. Это также означает, что по умолчанию будет использоваться последняя установленная версия. Это удобно еще и потому, что старая версия по-прежнему остается в файловой системе.

Ниже приводится содержимое вновь установленного файла */usr/local/bin/easy_install*:

```
#!/System/Library/Frameworks/Python.framework/Versions/2.5/Resources/
  Python.app/Contents/MacOS/Python
# EASY-INSTALL-ENTRY-SCRIPT:
'setuptools==0.6c8','console_scripts','easy_install'
__requires__ = 'setuptools==0.6c8'
import sys
from pkg_resources import load_entry_point

sys.exit(
load_entry_point('setuptools==0.6c8', 'console_scripts', 'easy_install')()
)
```

Главное здесь то, что при установке setuptools устанавливается сценарий с именем `easy_install`, который может использоваться для установки и управления программным кодом на языке Python. Второй по важности момент, ради которого мы привели содержимое сценария `easy_install`, заключается в том, что он относится к типу сценариев, которые создаются автоматически при использовании «точек входа», когда определяются пакеты. Пока не надо беспокоиться о содержимом этого сценария, о точках входа или о создании таких сценариев, как этот. Мы подойдем к этой теме далее в этой главе.

Теперь, когда в нашем распоряжении имеется сценарий `easy_install`, мы можем установить любой пакет, находящийся в центральном репозитарии модулей Python, который обычно называют PyPI (Python

Package Index – каталог пакетов Python), или «Cheesshop»: *http://pypi.python.org/pypi*.

Чтобы установить IPython, оболочку, которая используется для демонстрации примеров на протяжении всей книги, можно просто запустить следующую команду:

```
sudo easy_install ipython
```

Обратите внимание, что для выполнения своей работы сценарий `easy_install` требует в данном случае привилегий суперпользователя, так как пакеты устанавливаются в глобальный для Python каталог *site-packages*. Он также помещает сценарии в каталог, по умолчанию предназначенный операционной системой для сценариев, который обычно является каталогом, где находится исполняемый файл `python`. Для установки пакетов с помощью `easy_install` необходимо обладать правом на запись в каталог *site-packages* и в каталог, куда был установлен Python. Если у вас это вызывает затруднения, обратитесь к разделу главы, где обсуждается использование virtualenv и setuptools. Как вариант, можно было бы скомпилировать и установить Python в каталог по своему выбору: например, в свой домашний каталог.

Прежде чем мы перейдем к изучению дополнительных возможностей инструмента `easy_install`, коротко вспомним основные моменты использования `easy_install`:

1. Загрузить сценарий начальной установки `ez_setup.py`.
2. Запустить `ez_setup.py` для версии Python, с которой будет работать `easy_install`.
3. Если в вашей системе установлено несколько версий Python, явно запускайте сценарий `easy_install` с требуемым номером версии.

ПОРТРЕТ ЗНАМЕНИТОСТИ: EASY INSTALL

Филлип Дж. Эби (Phillip J. Eby)

Филлип Дж. Эби отвечает за систему Python Enhancement Proposals (система приема предложений по улучшению Python), поддержку стандарта WSGI (Web Server Gateway Interface – интерфейс взаимодействия с веб-сервером), setuptools и многое другое. О нем рассказывалось в книге «Dreaming in Code» (Three Rivers Press). Вы можете посетить его блог, посвященный программированию: *http://dirtsimple.org/programming/*.

Дополнительные особенности easy_install

Для большинства тех, кто пользуется сценарием easy_install, вполне достаточно вызывать его с единственным аргументом командной строки, без дополнительных параметров. (К слову сказать, при использовании easy_install с единственным аргументом – именем пакета, этот сценарий просто загрузит и установит этот пакет, как показано в предыдущем примере с IPython.) Тем не менее, иногда бывают случаи, когда неплохо было бы иметь возможности для выполнения дополнительных действий, помимо загрузки пакета из Python Package Index. К счастью, easy_install имеет в запасе несколько оригинальных решений и обладает достаточно высокой гибкостью, чтобы предоставить целый набор дополнительных возможностей.

Поиск пакетов на веб-страницах

Как было показано ранее, easy_install может отыскивать пакеты в центральном репозитарии и автоматически устанавливать их. Но кроме этого он имеет возможность устанавливать пакеты любыми другими способами, которые только можно себе представить. Ниже приводится пример того, как выполнить поиск пакета на веб-странице и установить или обновить пакет по имени и номеру версии:

```
$ easy_install -f http://code.google.com/p/liten/ liten
Searching for liten
Reading http://code.google.com/p/liten/
Best match: liten 0.1.3
Downloading http://liten.googlecode.com/files/liten-0.1.3-py2.4.egg
[обрезано]
```

В данной ситуации на странице *http://code.google.com/p/liten/* имеется пакет *.egg* для Python 2.4 и Python 2.5. Ключ -f сценария easy_install определяет адрес страницы, где требуется отыскать пакет. Сценарий отыскал оба пакета и установил версию для Python 2.4, как наиболее соответствующую. Достаточно очевидно, что это очень мощная особенность, так как easy_install не только отыскал ссылку на пакет, но и обнаружил наиболее подходящую версию.

Установка дистрибутива с исходными текстами по заданному URL

Теперь мы попробуем автоматически установить дистрибутив с исходными текстами по известному адресу URL:

```
% easy_install http://superb-west.dl.sourceforge.net/sourceforge
      /sqlalchemy/SQLAlchemy-0.4.3.tar.gz

Downloading http://superb-west.dl.sourceforge.net/sourceforge
/sqlalchemy/SQLAlchemy-0.4.3.tar.gz
Processing SQLAlchemy-0.4.3.tar.gz
```

```
Running SQLAlchemy-0.4.3/setup.py -q bdist_egg --dist-dir
  /var/folders/LZ/LZFo5h8JEW4Jzr+ydkXfI++++TI/-Tmp-/
  easy_install-Gw2Xq3/SQLAlchemy-0.4.3/egg-dist-tmp-Mf4jir
zip_safe flag not set; analyzing archive contents...
sqlalchemy.util: module MAY be using inspect.stack
sqlalchemy.databases.mysql: module MAY be using inspect.stack
Adding SQLAlchemy 0.4.3 to easy-install.pth file
Installed /Users/ngift/src/py24ENV/lib/python2.4/site-packages/SQLAlchemy-
0.4.3-py2.4.egg
Processing dependencies for SQLAlchemy==0.4.3
Finished processing dependencies for SQLAlchemy==0.4.3
```

Мы передали сценарию `easy_install` адрес местоположения сжатого тарболла. Он обнаружил, что должен установить дистрибутив с исходными текстами, причем нам не пришлось явно сообщать ему об этом. Это очень интересный способ установки, но, чтобы он работал, в корневом каталоге дистрибутива должен иметься файл *setup.py*. Например, к моменту написания этих строк, если разработчик создаст несколько уровней вложенности каталогов, попытка выполнить установку такого пакета потерпит неудачу.

Установка пакетов, расположенных в локальной или в сетевой файловой системе

Ниже приводится пример установки пакета *.egg*, находящегося в локальной файловой системе или на смонтированном томе NFS:

```
easy_install /net/src/eggs/convertWindowsToMacOperatingSystem-py2.5.egg
```

Кроме всего прочего существует возможность устанавливать пакеты, находящиеся в смонтированном каталоге NFS или в локальном разделе. Это может быть очень удобно при распространении пакетов в окружении *nix, особенно когда имеется несколько машин, которые должны быть синхронизированы друг с другом по версиям программного кода, работающего на них. Некоторые другие сценарии, представленные в этой книге, могли бы помочь в создании демона, выполняющего опрос. Каждый клиент мог бы с помощью такого демона проверять наличие обновлений в центральном репозитарии пакетов. При обнаружении новой версии он мог бы автоматически выполнять обновление.

Обновление пакетов

Еще одна область применения `easy_install` – обновление пакетов. В следующих нескольких примерах демонстрируется установка и обновление пакета CherryPy.

Сначала устанавливается версия CherryPy 2.2.1:

```
$ easy_install cherrypy==2.2.1
Searching for cherrypy==2.2.1
Reading http://pypi.python.org/simple/cherrypy/
```

```
....
Best match: CherryPy 2.2.1
Downloading http://download.cherrypy.org/cherrypy/2.2.1/CherryPy-2.2.1.tar.gz
....
Processing dependencies for cherrypy==2.2.1
Finished processing dependencies for cherrypy==2.2.1
```

Теперь посмотрим, что произойдет, если предложить сценарию `easy_install` попытаться установить пакет, который уже был установлен:

```
$ easy_install cherrypy
Searching for cherrypy
Best match: CherryPy 2.2.1
Processing CherryPy-2.2.1-py2.5.egg
CherryPy 2.2.1 is already the active version in easy-install.pth

Using /Users/jmjones/python/cherrypy/lib/python2.5/site-packages/CherryPy-
2.2.1-py2.5.egg
Processing dependencies for cherrypy
Finished processing dependencies for cherrypy
```

После установки некоторой версии пакета можно обновить его до более свежей версии, явно указав, какую версию нужно загрузить и установить:

```
$ easy_install cherrypy==2.3.0 Searching for
cherrypy==2.3.0
Reading http://pypi.python.org/simple/cherrypy/
....
Best match: CherryPy 2.3.0
Downloading http://download.cherrypy.org/cherrypy/2.3.0/CherryPy-2.3.0.zip
....
Processing dependencies for cherrypy==2.3.0
Finished processing dependencies for cherrypy==2.3.0
```

Обратите внимание: в данном примере мы не использовали ключ `--upgrade`. В действительности этот ключ необходим, только если у вас уже установлена некоторая версия пакета и вы хотите обновить его до самой последней версии.

Далее мы обновляем пакет до версии CherryPy 3.0.0, используя ключ `--upgrade`. В данном случае использовать ключ `--upgrade` было совершенно необязательно:

```
$ easy_install --upgrade cherrypy==3.0.0
Searching for cherrypy==3.0.0
Reading http://pypi.python.org/simple/cherrypy/
....
Best match: CherryPy 3.0.0
Downloading http://download.cherrypy.org/cherrypy/3.0.0/CherryPy-3.0.0.zip
....
Processing dependencies for cherrypy==3.0.0
Finished processing dependencies for cherrypy==3.0.0
```

Если использовать ключ `--upgrade`, не указывая номер версии, обновление будет выполнено до самой последней версии пакета. Обратите внимание: действие команды в этом случае отличается от действия команды `easy_install cherrypy`. Команда `easy_install cherrypy` обнаружит, что ранее уже была установлена некоторая версия пакета и поэтому никаких действий предпринимать не будет. В следующем примере произойдет обновление пакета CherryPy до самой последней версии:

```
$ easy_install --upgrade cherrypy
Searching for cherrypy
Reading http://pypi.python.org/simple/cherrypy/
....
Best match: CherryPy 3.1.0beta3
Downloading http://download.cherrypy.org/cherrypy/3.1.0beta3/CherryPy-
3.1.0beta3.zip
....
Processing dependencies for cherrypy
Finished processing dependencies for cherrypy
```

Теперь у нас установлена версия CherryPy 3.1.0b3. Если теперь попробовать выполнить обновление до версии, больше чем 3.0.0, никаких действий предприниматься не будет, так как у нас уже установлена такая версия:

```
$ easy_install --upgrade cherrypy>3.0.0
$
```

Установка распакованного дистрибутива с исходными текстами в текущем рабочем каталоге

Несмотря на всю свою тривиальность, такой способ установки может оказаться полезным. Вместо того чтобы следовать через процедуру `python setup.py install`, вы можете просто ввести следующую команду (для этого требуется меньше вводить с клавиатуры, поэтому это будет ценный совет для ленивых):

```
easy_install
```

Извлечение дистрибутива с исходными текстами в заданный каталог

Следующий пример может использоваться для поиска дистрибутива с исходными текстами или пакета по указанному URL с последующей распаковкой его в заданный каталог:

```
easy_install --editable --build-directory ~/sandbox liten
```

Это достаточно удобный прием, так как он позволяет с помощью `easy_install` поместить дистрибутив с исходными текстами в требуемый каталог. Так как в процессе установки пакета с помощью сценария `easy_install` не всегда устанавливается все его содержимое (например,

документация или примеры программного кода), это отличный способ узнать, что входит в состав дистрибутива. В этом случае easy_install только лишь скопирует файлы из дистрибутива. Если вам потребуется установить пакет, вам нужно будет запустить сценарий easy_install еще раз.

Изменение активной версии пакета

В этом примере предполагается, что у вас уже установлен пакет liten версии 0.1.3 и при этом была установлена некоторая другая версия liten. Кроме того, предполагается, что «активной» является эта другая версия. Ниже показано, как можно активировать версию:

```
easy_install liten=0.1.3
```

Этот прием работает как в случае перехода к использованию более старой версии, так и в случае возврата к более новой версии пакета.

Преобразование отдельного файла .py в пакет .egg

Ниже показано, как преобразовать обычный пакет Python в пакет *.egg* (обратите внимание на ключ -f):

```
easy_install -f "http://svn.colorstudy.com/virtualenv/
      trunk/virtualenv.py#egg=virtualenv-1.0" virtualenv
```

Этот прием пригодится, когда необходимо упаковать единственный файл *.py* в пакет *.egg*. Иногда этот метод может оказаться лучшим способом, когда необходимо обеспечить доступ к ранее распакованному отдельному файлу из любой точки файловой системы. Как вариант, можно было бы добавить путь к требуемому файлу в переменную окружения PYTHONPATH. В этом примере мы получаем из основного каталога проекта сценарий virtualenv.py, упаковываем его и указываем нашу собственную версию и метку к ней. В строке URL подстрока #egg=virtualenv-1.0 просто определяет имя пакета и номер версии, выбранные нами для этого сценария. Аргумент, который следует за строкой URL, определяет имя создаваемого пакета. В этом аргументе желательно использовать имена, которые не противоречили бы строке URL, потому что мы предписываем easy_install установить пакет с тем же именем, что и созданный. Даже при том, что было бы желательно указывать непротиворечивые имена, вы не должны чувствовать себя обязанными сохранять название пакета в соответствии с названием модуля. Например:

```
easy_install -f "http://svn.colorstudy.com/virtualenv/
      trunk/virtualenv.py#egg=foofoo-1.0" foofoo
```

Эта команда делает в точности то же самое, что и предыдущий пример, только в этом случае создается пакет с именем foofoo, а не virtualenv. Какое имя вы выберете для своего пакета – полностью ваше дело.

Аутентификация на сайтах, доступ к которым защищен паролем

В практике может возникнуть ситуация, когда вам потребуется установить пакет *.egg* с веб-сайта, где требуется пройти аутентификацию, прежде чем будет разрешено загружать с него какие-либо файлы. В таком случае можно указать имя пользователя и пароль прямо в строке URL, как показано ниже:

```
easy_install -f http://uid:passwd@example.com/packages
```

Вы можете заниматься на работе своим собственным проектом, и вам не хотелось бы, чтобы ваши коллеги узнали о нем. (Разве не все занимаются этим?) Один из способов передать свои пакеты коллегам «из-под полы» заключается в том, чтобы создать простой файл *.htaccess* и затем выполнять процедуру аутентификации в сценарии `easy_install`.

Использование конфигурационных файлов

Для опытных пользователей в арсенале `easy_install` имеется еще один прием. Значения параметров по умолчанию можно задать с помощью конфигурационного файла, который имеет формат ini-файлов. Для системного администратора это особенно удачная возможность, так как позволяет определять настройки клиентов, использующих сценарий `easy_install`. Параметры конфигурации будут отыскиваться сценарием `easy_install` в следующих файлах и в следующем порядке: *текущий_рабочий_каталог/setup.cfg*, *~/.pydistutils.cfg* и в файле *distutils.cfg*, в каталоге пакета distutils.

Что можно добавить в этот конфигурационный файл? Обычно здесь определяются два параметра: сайт(ы) в локальной сети, откуда можно загружать пакеты, и нестандартный каталог установки пакетов. Ниже показано, как может выглядеть файл конфигурации для `easy_install`:

```
[easy_install]

#Где искать пакеты
find_links = http://code.example.com/downloads

#Ограничить возможности поиска этими доменами
allow_hosts = *.example.com

#Куда устанавливать пакеты. Обратите внимание: этот каталог должен
#находиться в PYTHONPATH
install_dir = /src/lib/python
```

В этом конфигурационном файле, который мы могли бы назвать, например, *~/.pydistutils.cfg*, определяется адрес URL для поиска пакетов, разрешается искать пакеты только в домене *example.com* (и в поддоменах) и, наконец, указывается, куда должны помещаться пакеты при установке.

Коротко о дополнительных особенностях easy_install

Этот раздел не может служить заменой всеобъемлющей официальной документации с описанием сценария `easy_install`, его цель состояла лишь в том, чтобы обозначить некоторые основные особенности, которые могут использоваться опытными пользователями. Сценарий `easy_install` продолжает активно разрабатываться, поэтому будет нелишним чаще обращаться на страницу *http://peak.telecommunity.com/DevCenter/EasyInstall* за обновлениями в документации. Кроме того, существует почтовая рассылка, она называется distutils-sig (где sig происходит от special interest group – группа с особыми интересами), где обсуждаются все проблемы, связанные с распространением программного кода. Подпишитесь на рассылку по адресу *http://mail.python.org/mailman/listinfo/distutils-sig*, и вы сможете посылать свои сообщения об обнаруженных ошибках и получать помощь, касающуюся использования `easy_install`.

Наконец, выполнив команду `easy_install --help`, вы обнаружите еще большее число параметров, о которых мы даже не упоминали здесь. Весьма вероятно, что какая-либо особенность, которая вам необходима, уже реализована в `easy_install`.

Создание пакетов

Ранее мы уже упоминали, что пакеты с расширением *.egg* – это пакеты модулей на языке Python, но мы не давали определения пакетам лучше, чем это. Ниже приводится определение «пакета egg», взятое с вебсайта проекта setuptools:

> Пакеты в формате *.egg* – это предпочтительный двоичный формат дистрибутивов для EasyInstall, потому что являются кросс-платформенными (для пакетов с модулями исключительно на языке Python), могут импортироваться непосредственно и содержат метаданные проекта, включая сценарии и информацию о зависимостях проекта. Они могут просто загружаться и добавляться в значение атрибута sys.path или помещаться в каталог, который уже имеется в sys.path, после чего они автоматически будут обнаруживаться системой управления пакетами во время выполнения.

Мы не приводили причины, по которым системный администратор мог бы быть заинтересован в создании пакетов. Если ваша деятельность ограничивается созданием одноразовых сценариев, эти знания не окажутся вам особенно полезными. Но когда вы начнете выделять общие шаблоны и задачи, вы обнаружите, что пакеты помогут вам избежать многих неприятностей. Если вы создадите небольшую библиотеку сценариев, предназначенных для решения наиболее часто встречающихся задач, вы сможете оформить ее в виде пакета. А если вы сделаете это, то вы не только сэкономите время на написании про-

граммного кода, но облегчите себе процедуру их установки на нескольких машинах.

Создание пакетов Python представляет собой чрезвычайно простой процесс, состоящий из четырех этапов:

1. Установить setuptools.
2. Создать файлы, которые необходимо поместить в пакет.
3. Создать файл *setup.py*.
4. Запустить.

```
python setup.py bdist_egg
```

Инструмент `setuptools` у нас уже установлен, поэтому мы можем двинуться дальше и создать файлы для помещения в пакет:

```
$ cd /tmp
$ mkdir egg-example
$ cd egg-example
$ touch hello-egg.py
```

В данном случае пакет будет содержать пустой модуль на языке Python с именем `hello-egg.py`.

Далее создадим простейший файл *setup.py*:

```
from setuptools import setup, find_packages
setup(
    name = "HelloWorld",
    version = "0.1",
    packages = find_packages(),
)
```

Теперь создадим пакет:

```
$ python setup.py bdist_egg
running bdist_egg
running egg_info
creating HelloWorld.egg-info
writing HelloWorld.egg-info/PKG-INFO
writing top-level names to HelloWorld.egg-info/top_level.txt
writing dependency_links to HelloWorld.egg-info/dependency_links.txt
writing manifest file 'HelloWorld.egg-info/SOURCES.txt'
reading manifest file 'HelloWorld.egg-info/SOURCES.txt'
writing manifest file 'HelloWorld.egg-info/SOURCES.txt'
installing library code to build/bdist.macosx-10.5-i386/egg
running install_lib
warning: install_lib: 'build/lib' does not exist -- no Python modules to
install
creating build
creating build/bdist.macosx-10.5-i386
creating build/bdist.macosx-10.5-i386/egg
creating build/bdist.macosx-10.5-i386/egg/EGG-INFO
```

```
copying HelloWorld.egg-info/PKG-INFO -> build/bdist.macosx-10.5-i386/egg/
EGG-INFO
copying HelloWorld.egg-info/SOURCES.txt -> build/bdist.macosx-10.5-i386/egg/
EGG-INFO
copying HelloWorld.egg-info/dependency_links.txt -> build/bdist.macosx-10.5-
i386/egg/EGG-INFO
copying HelloWorld.egg-info/top_level.txt -> build/bdist.macosx-10.5-i386/
egg/EGG-INFO
zip_safe flag not set; analyzing archive contents...
creating dist
creating 'dist/HelloWorld-0.1-py2.5.egg' and adding 'build/bdist.macosx-
10.5-i386/egg' to it
removing 'build/bdist.macosx-10.5-i386/egg' (and everything under it)
$ ll
total 8
drwxr-xr-x  6 ngift  wheel  204 Mar 10 06:53 HelloWorld.egg-info
drwxr-xr-x  3 ngift  wheel  102 Mar 10 06:53 build
drwxr-xr-x  3 ngift  wheel  102 Mar 10 06:53 dist
-rw-r--r--  1 ngift  wheel    0 Mar 10 06:50 hello-egg.py
-rw-r--r--  1 ngift  wheel  131 Mar 10 06:52 setup.py
```

Установим пакет:

```
$ sudo easy_install HelloWorld-0.1-py2.5.egg
sudo easy_install HelloWorld-0.1-py2.5.egg
Password:
Processing HelloWorld-0.1-py2.5.egg
Removing /Library/Python/2.5/site-packages/HelloWorld-0.1-py2.5.egg
Copying HelloWorld-0.1-py2.5.egg to /Library/Python/2.5/site-packages
Adding HelloWorld 0.1 to easy-install.pth file

Installed /Library/Python/2.5/site-packages/HelloWorld-0.1-py2.5.egg
Processing dependencies for HelloWorld==0.1
Finished processing dependencies for HelloWorld==0.1
```

Как видите, пакеты создаются чрезвычайно просто. Однако созданный пакет в действительности был пустым файлом, поэтому мы создадим другой сценарий на языке Python и рассмотрим процедуру создания пакета более подробно.

Ниже приводится очень простой сценарий на языке Python, который отображает файлы в каталоге, являющиеся символическими ссылками, показывает местоположение, где находятся соответствующие им настоящие файлы, и выясняет, существуют ли эти файлы на самом деле:

```
#!/usr/bin/env python

import os
import sys

def get_dir_tuple(filename, directory):
    abspath = os.path.join(directory, filename)
    realpath = os.path.realpath(abspath)
    exists = os.path.exists(abspath)
```

```
    return (filename, realpath, exists)

def get_links(directory):
    file_list = [get_dir_tuple(f, directory) for f in os.listdir(directory)
                 if os.path.islink(os.path.join(directory, f))]
    return file_list

def main():
    if not len(sys.argv) == 2:
        print 'USAGE: %s directory' % sys.argv[0]
        sys.exit(1)
    directory = sys.argv[1]
    print get_links(directory)

if __name__ == '__main__':
    main()
```

Затем мы создадим сценарий `setup.py`, который будет использоваться инструментом setuptools. Это еще один минимально возможный файл *setup.py*, как и в предыдущем примере:

```
from setuptools import setup, find_packages
setup(
    name = "symlinkator",
    version = "0.1",
    packages = find_packages(),
    entry_points = {
        'console_scripts': [
            'linkator = symlinkator.symlinkator:main',
        ],
    },
)
```

Здесь объявляется имя пакета – «symlinkator», номер версии 0.1 и указывается, что инструмент setuptools будет пытаться отыскать любые подходящие файлы для включения. Раздел `entry_points` пока просто игнорируйте.

Теперь соберем пакет, запустив команду `python setup.py bdist_egg`:

```
$ python setup.py bdist_egg
running bdist_egg
running egg_info
creating symlinkator.egg-info
writing symlinkator.egg-info/PKG-INFO
writing top-level names to symlinkator.egg-info/top_level.txt
writing dependency_links to symlinkator.egg-info/dependency_links.txt
writing manifest file 'symlinkator.egg-info/SOURCES.txt'
writing manifest file 'symlinkator.egg-info/SOURCES.txt'
installing library code to build/bdist.linux-x86_64/egg
running install_lib
warning: install_lib: 'build/lib' does not exist -- no Python modules
to install
creating build
```

```
creating build/bdist.linux-x86_64
creating build/bdist.linux-x86_64/egg
creating build/bdist.linux-x86_64/egg/EGG-INFO
copying symlinkator.egg-info/PKG-INFO -> build/bdist.linux-x86_64/egg/EGG-
INFO
copying symlinkator.egg-info/SOURCES.txt -> build/bdist.linux-x86_64/egg/
EGG-INFO
copying symlinkator.egg-info/dependency_links.txt -> build/bdist.linux-
x86_64/egg/EGG-INFO
copying symlinkator.egg-info/top_level.txt -> build/bdist.linux-x86_64/egg/
EGG-INFO
zip_safe flag not set; analyzing archive contents...
creating dist
creating 'dist/symlinkator-0.1-py2.5.egg' and adding 'build/bdist.linux-
x86_64/egg' to it
removing 'build/bdist.linux-x86_64/egg' (and everything under it)
```

Проверим содержимое пакета. Для этого перейдем в предварительно созданный каталог *dist* и проверим наличие пакета в этом каталоге:

```
$ ls -l dist
total 4
-rw-r--r-- 1 jmjones jmjones 825 2008-05-03 15:34 symlinkator-0.1-py2.5.egg
```

Теперь установим пакет:

```
$ easy_install dist/symlinkator-0.1-py2.5.egg
Processing symlinkator-0.1-py2.5.egg
....
Processing dependencies for symlinkator==0.1
Finished processing dependencies for symlinkator==0.1
```

Затем запустим оболочку IPython, импортируем модуль и попробуем им воспользоваться:

```
In [1]: from symlinkator.symlinkator import get_links

In [2]: get_links('/home/jmjones/logs/')
Out[2]: [('fetchmail.log.old', '/home/jmjones/logs/fetchmail.log.3', False),
 ('fetchmail.log', '/home/jmjones/logs/fetchmail.log.0', True)]
```

На всякий случай проверим содержимое каталога, который был передан функции `get_links()`:

```
$ ls -l ~/logs/
total 0
lrwxrwxrwx 1 jmjones jmjones 15 2008-05-03 15:11 fetchmail.log ->
fetchmail.log.0
-rw-r--r-- 1 jmjones jmjones  0 2008-05-03 15:09 fetchmail.log.0
-rw-r--r-- 1 jmjones jmjones  0 2008-05-03 15:09 fetchmail.log.1
lrwxrwxrwx 1 jmjones jmjones 15 2008-05-03 15:11 fetchmail.log.old ->
fetchmail.log.3
```

Точки входа и сценарии консоли

Со страницы документации проекта setuptools:

> Точки входа используются для поддержки динамического обнаружения служб или расширений, предоставляемых проектом. За дополнительной информацией и примерами представления аргументов обращайтесь к разделу «Dynamic Discovery of Services and Plugins». Кроме того, это ключевое слово (entry_points) используется для поддержки автоматического создания сценариев.

В этой книге мы рассмотрим единственную разновидность точек входа – различные сценарии консоли. setuptools автоматически создает сценарий консоли, исходя из двух частей информации, которую вы поместите в свой сценарий *setup.py*. Ниже приводится интересующий нас раздел в файле *setup.py* из предыдущего примера:

```
entry_points = {
    'console_scripts': [
        'linkator = symlinkator.symlinkator:main',
    ],
},
```

В этом примере мы указали, что хотели бы получить сценарий `linkator` и что при исполнении сценария он должен вызывать функцию `main()` из модуля `symlinkator.symlinkator`. Во время установки пакета сценарий `linkator` был помещен в тот же каталог, где находится исполняемый файл `python`:

```
#!/home/jmjones/local/python/scratch/bin/python
# EASY-INSTALL-ENTRY-SCRIPT: 'symlinkator==0.1','console_scripts','linkator'
__requires__ = 'symlinkator==0.1'
import sys
from pkg_resources import load_entry_point

sys.exit(
    load_entry_point('symlinkator==0.1', 'console_scripts', 'linkator')()
)
```

Все, что вы видите, было создано инструментом `setuptools`. Совершенно необязательно понимать все, что находится в этом сценарии. В действительности, вообще необязательно понимать хоть что-нибудь в этом сценарии. Важно лишь знать, что, когда вы определяете в файле `setup.py` точку входа `console_scripts`, setuptools создаст сценарий, который будет вызывать ваш программный код, расположенный там, где вы укажете. Ниже показано, что произошло, когда мы вызвали этот сценарий примерно так, как вызывали функцию в предыдущем примере:

```
$ linkator ~/logs/
[('fetchmail.log.old', '/home/jmjones/logs/fetchmail.log.3', False),
('fetchmail.log', '/home/jmjones/logs/fetchmail.log.0', True)]
```

С точками входа связано несколько достаточно сложных аспектов, но на верхнем уровне достаточно знать, что точки входа используются для установки ваших сценариев, играющих роль инструментов командной строки, в каталог, доступный для пользователя. Для этого вам необходимо следовать синтаксису, описанному выше, и определить функцию, которая должна вызываться вашим инструментом командной строки.

Регистрация пакета в Python Package Index

Если вы напишете действительно полезный модуль, вполне естественно, что вы захотите поделиться им с другими людьми. Это одна из самых приятных сторон в разработке открытого программного обеспечения. К счастью, выгрузить пакет в Python Package Index (каталог пакетов Python) совсем несложно.

Этот процесс лишь немного отличается от процесса создания пакета. При этом следует обратить внимание на две вещи: не забыть включить описание в формате ReST (reStructuredText) в атрибут `long_description` и подставить значение `download_url`. О формате ReST мы уже упоминали в главе 4.

Несмотря на то, что формат ReST уже обсуждался ранее, мы должны здесь подчеркнуть, что применение формата ReST для оформления документации необходимо потому, что она будет преобразована в формат HTML после выгрузки пакета в Python Package Index. Вы можете воспользоваться инструментом ReSTless, созданным Аароном Хиллегассом (Aaron Hillegass), для предварительного просмотра форматированного текста, чтобы убедиться, что он отформатирован должным образом. Обязательно просматривайте документацию, чтобы убедиться в отсутствии нарушений форматирования. Если текст не будет должным образом отформатирован в формате ReST, после выгрузки модуля текст будет отображаться как обычный текст, а не как HTML.

В примере 9.2 приводится содержимое файла `setup.py` для инструмента командной строки и библиотеки, созданной Ноа (Noah).

Пример 9.2. Пример файла setup.py для выгрузки модуля в Python Package Index

```
#!/usr/bin/env python

# liten 0.1.4.2 -- deduplication command-line tool
#
# Author: Noah Gift
try:
    from setuptools import setup, find_packages
except ImportError:
    from ez_setup import use_setuptools
    use_setuptools()
    from setuptools import setup, find_packages
```

```
import os,sys

version = '0.1.4.2'
f = open(os.path.join(os.path.dirname(__file__), 'docs', 'index.txt'))
long_description = f.read().strip()
f.close()

setup(
    name='liten',
    version='0.1.4.2',
    description='a de-duplication command line tool',
    long_description=long_description,
    classifiers=[
        'Development Status :: 4 - Beta',
        'Intended Audience :: Developers',
        'License :: OSI Approved :: MIT License',
        ],
    author='Noah Gift',
    author_email='noah.gift@gmail.com',
    url='http://pypi.python.org/pypi/liten',
    download_url="http://code.google.com/p/liten/downloads/list",
    license='MIT',
    py_modules=['virtualenv'],
    zip_safe=False,
    py_modules=['liten'],
    entry_points="""
    [console_scripts]
    liten = liten:main
    """,
)
```

С помощью этого файла *setup.py* теперь можно «автоматически» зарегистрировать пакет в Python Package Index, используя следующую команду:

```
$ python setup.py register
running register
running egg_info
writing liten.egg-info/PKG-INFO
writing top-level names to liten.egg-info/top_level.txt
writing dependency_links to liten.egg-info/dependency_links.txt
writing entry points to liten.egg-info/entry_points.txt
writing manifest file 'liten.egg-info/SOURCES.txt'
Using PyPI login from /Users/ngift/.pypirc
Server response (200): OK
```

В этом файле `setup.py` появились новые дополнительные поля, если сравнивать его с предыдущим примером `symlinkator`. В число дополнительных полей входят `description`, `long_description`, `classifiers`, `author` и `download_url`. Точка входа, обсуждавшаяся выше, позволяет запускать инструмент из командной строки и устанавливать его в каталог, по умолчанию предназначенный для сценариев.

Атрибут `download_url` имеет особо важное значение, потому что он сообщает сценарию `easy_install`, где искать ваш пакет. Сюда можно включить ссылку на страницу, и тогда сценарий `easy_install` самостоятельно отыщет дистрибутив с исходными текстами или пакет в формате *.egg*, но можно также указать прямую ссылку на пакет.

В атрибут `long_description` записывается существующее описание, которое хранится в созданном нами файле *index.txt* в подкаталоге *docs*. Файл *index.txt* содержит текст в формате ReST, а сценарий `setup.py` в процессе регистрации пакета в Python Package Index читает эту информацию и помещает ее в атрибут `long_description`.

Где можно получить дополнительную информацию...

Ниже приводится несколько важных ресурсов:

Easy install

http://peak.telecommunity.com/DevCenter/EasyInstall

Пакеты Python в формате .egg

http://peak.telecommunity.com/DevCenter/PythonEggs

Модуль setuptools

http://peak.telecommunity.com/DevCenter/setuptools

Модуль pkg_resources

http://peak.telecommunity.com/DevCenter/PkgResources

Обзор архитектуры модуля pkg_resources и формата Python egg в общих чертах

Обзор архитектуры модуля pkg_resources и формата Python egg в общих чертах

И не забудьте про почтовую рассылку по языку Python *http://mail.python.org/pipermail/distutilssig/*.

Distutils

К моменту написания этих строк инструмент setuptools считался предпочтительным способом создания и распространения пакетов и похоже, что части библиотеки setuptools войдут в стандартную библиотеку языка Python. Но при этом по-прежнему важно знать, как работает пакет distutils, возможности которого расширяет библиотека setuptools, и какие возможности в нем отсутствуют.

Когда пакет создается с помощью distutils, обычно установка такого пакета выполняется командой:

```
python setup.py install
```

Что касается вопроса сборки пакетов, готовых для распространения, мы рассмотрим четыре следующие темы:

- Как написать сценарий установки, то есть файл *setup.py*
- Основные параметры настройки в файле *setup.py*
- Как собрать дистрибутив с исходными текстами
- Создание двоичных пакетов, например, в формате rpm для Red Hat, pkgtool для Solaris и swinstall для HP-UX

Лучший способ продемонстрировать, как работает пакет distutils, – это встать на ноги и приготовиться.

Шаг 1: создать некоторый программный код. Воспользуемся следующим сценарием, на примере которого продемонстрируем, как организовать его распространение:

```
#!/usr/bin/env python
#A simple python script we will package
#Distutils Example. Version 0.1

class DistutilsClass(object):
    """Этот класс выводит информацию о самом себе."""

    def __init__(self):
        print "Hello, I am a distutils distributed script." \
            "All I do is print this message."

if __name__ == '__main__':
    DistutilsClass()
```

Шаг 2: создать файл *setup.py* в каталоге со сценарием.

```
#Installer for distutils example script

from distutils.core import setup

setup(name="distutils_example",
    version="0.1",
    description="A Completely Useless Script That Prints",
    author="Joe Blow",
    author_email = "joe.blow@pyatl.org",
    url = "http://www.pyatl.org")
```

Обратите внимание: мы передали функции `setup()` несколько именованных аргументов, значения которых позднее будут использоваться как метаданные для идентификации пакета. Учтите, что это очень простой пример и на самом деле эта функция имеет гораздо большее число аргументов, которые, например, позволяют разрешать проблемы с зависимостями и другие. Мы не будем углубляться в исследование дополнительных параметров настройки, но рекомендуем ознакомиться с ними в официальной электронной документации Python.

Шаг 3: создать дистрибутив.

Теперь, когда у нас имеется простенький сценарий `setup.py`, можно создать пакет дистрибутива с исходными текстами, просто запустив следующую команду в том же каталоге, где находится ваш сценарий и файлы *README* и *setup.py*:

```
python setup.py sdist
```

Вы должны получить следующий вывод:

```
running sdist
    warning: sdist: manifest template 'MANIFEST.in' does not exist
      (using default file list)
    writing manifest file 'MANIFEST'
    creating distutils_example-0.1
    making hard links in distutils_example-0.1...
    hard linking README.txt distutils_example-0.1
    hard linking setup.py distutils_example-0.1
    creating dist
    tar -cf dist/distutils_example-0.1.tar distutils_example-0.1
    gzip -f9 dist/distutils_example-0.1.tar
    removing 'distutils_example-0.1' (and everything under it)
```

Теперь все, что необходимо сделать для установки такого пакета, – это распаковать его и выполнить команду:

```
python setup.py install
```

Ниже приводятся несколько примеров, которые пригодятся, если возникнет необходимость создать пакет в двоичном формате. Обратите внимание, что эти способы тесно связаны с типом операционной системы, поэтому вы не сможете собрать пакет в формате rpm, например, в операционной системе OS X. Однако, учитывая изобилие продуктов виртуализации, это не должно быть для вас большой проблемой. Достаточно хранить под рукой несколько виртуальных машин, которыми можно было бы воспользоваться для сборки пакетов.

Чтобы собрать пакет rpm:

```
python setup.py bdist_rpm
```

Чтобы собрать пакет в формате pkgtool для Solaris:

```
python setup.py bdist_pkgtool
```

Чтобы собрать пакет в формате swinstall для HP-UX:

```
python setup.py bdist_sdux
```

Наконец, когда вы выполняете компиляцию полученного пакета, вы можете настроить каталог, где должна происходить сборка. Обычно процессы компиляции и установки выполняются одновременно, но вы можете выбрать для сборки свой каталог, как показано ниже:

```
python setup.py build --build-base=/mnt/python_src/ascript.py
```

Когда вы запустите команду `install`, она скопирует все, что находится в каталоге *сборки*, в каталог *установки*. По умолчанию каталог установки соответствует каталогу *site-packages* в каталоге установки интерпретатора Python, под управлением которого выполняется коман-

да, но вы можете указать свой каталог *установки*, например, смонтированный каталог NFS, как в примере выше.

Buildout

Инструмент Buildout был создан Джимом Фултоном (Jim Fulton) из корпорации Zope Corporation и предназначен для «сборки» новых приложений. Этими приложениями могут быть программы на языке Python или другие программы, такие как Apache. Одна из основных целей Buildout состоит в том, чтобы обеспечить возможность установки приложений на разных платформах. Одним из первых экспериментов, которые провел автор с помощью Buildout, был эксперимент по развертыванию сайта Plone 3.x. С тех пор он понял, что это была всего лишь вершина айсберга.

Buildout – это один из наиболее интересных новых инструментов управления пакетами, которые может предложить Python, так как он позволяет сложным приложениям со сложными зависимостями развертывать самих себя, если в дистрибутиве имеются файл *bootstrap.py* и файл с настройками. В следующих разделах мы поделим наше обсуждение на две части: использование Buildout и разработка с применением Buildout. Мы также рекомендовали бы вам прочитать руководство по Buildout по адресу: *http://pypi.python.org/pypi/zc.buildout*, так как этот неоценимый ресурс содержит самую свежую информацию о Buildout. Эта документация настолько полная, насколько это возможно, и ее обязательно должен прочитать каждый пользователь Buildout.

ПОРТРЕТ ЗНАМЕНИТОСТИ: BUILDOUT

Джим Фултон (Jim Fulton)

Джим Фултон – создатель и один из членов проекта Zope Object Database. Джим также является одним из создателей Zope Object Publishing Environment и техническим директором Zope Corporation.

Использование Buildout

Несмотря на то, что многие, кто имел дело с технологиями Zope, знали о существовании Buildout, это оставалось тайной для остальной части пользователей Python. Buildout – это рекомендуемый механизм развертывания Plone. Для тех, кто не знаком с Plone: это система управ-

ления содержимым для сайтов уровня предприятия, за которой стоит огромное сообщество разработчиков. Система Plone была чрезвычайно сложна в установке, пока не появился инструмент Buildout. Теперь благодаря Buildout установка Plone выполняется тривиально просто.

Многие даже не подозревают, что Buildout можно использовать даже для управления окружением Python. Buildout – это весьма интеллектуальное программное обеспечение, которому требуется всего две вещи:

- Последняя версия *bootstrap.py*. Загрузить ее всегда можно по адресу *http://svn.zope.org/*checkout*/zc.buildout/trunk/bootstrap/bootstrap.py*.
- Файл *buildout.cfg* с именами пакетов для установки.

Лучший способ продемонстрировать возможности Buildout состоит в том, чтобы установить что-нибудь с его помощью. Ноа (Noah) написал инструмент командной строки для удаления дубликатов файлов, который можно найти в центральном репозитарии Python, PyPI. Мы попробуем с помощью Buildout развернуть среду Python для запуска этого инструмента.

Шаг 1: загруить сценарий buildout.py:

```
mkdir -p ~/src/buildout_demo
curl http://svn.zope.org/*checkout*/zc.buildout/trunk/
bootstrap/bootstrap.py > ~/src/buildout_demo/bootstrap.py
```

Шаг 2: написать простой файл *buildout.cfg*. Как уже говорилось выше, Buildout требует для своей работы файл *buildout.cfg*. Если попытаться запустить сценарий `buildout.py` без файла *buildout.cfg*, будет получено следующее сообщение:

```
$ python bootstrap.py
While:
Initializing.
Error: Couldn't open /Users/ngift/src/buildout_demo/buildout.cfg
```

Для примера создадим конфигурационный файл, как показано в примере 9.3.

Пример 9.3. Пример конфигурационного файла Buildout

```
[buildout]
parts = mypython
[mypython]
recipe = zc.recipe.egg
interpreter = mypython
eggs = liten
```

Если сохранить этот файл с именем `buildout.cfg` и затем снова запустить сценарий `buildout.py`, будет получен вывод, как показано в примере 9.4.

Пример 9.4. Создание окружения buildout

```
$ python bootstrap.py
Creating directory '/Users/ngift/src/buildout_demo/bin'.
Creating directory '/Users/ngift/src/buildout_demo/parts'.
Creating directory '/Users/ngift/src/buildout_demo/eggs'.
Creating directory '/Users/ngift/src/buildout_demo/develop-eggs'.
Generated script '/Users/ngift/src/buildout_demo/bin/buildout'.
```

Если заглянуть в эти вновь созданные каталоги, мы найдем выполняемые программы, включая отдельный интерпретатор Python в каталоге *bin*:

```
$ ls -l bin
total 24
-rwxr-xr-x 1 ngift staff 362 Mar 4 22:17 buildout
-rwxr-xr-x 1 ngift staff 651 Mar 4 22:23 mypython
```

Теперь, когда была выполнена установка инструмента Buildout, можно запустить его и пакет, который мы определили ранее, будет работать, как показано в примере 9.5.

Пример 9.5. Запуск Buildout и тестирование установки

```
$ bin/buildout
Getting distribution for 'zc.recipe.egg'.
Got zc.recipe.egg 1.0.0.
Installing mypython.
Getting distribution for 'liten'.
Got liten 0.1.3.
Generated script '/Users/ngift/src/buildout_demo/bin/liten'.
Generated interpreter '/Users/ngift/src/buildout_demo/bin/mypython'.
$ bin/mypython

>>>
$ ls -l bin
total 24
-rwxr-xr-x 1 ngift staff 362 Mar 4 22:17 buildout
-rwxr-xr-x 1 ngift staff 258 Mar 4 22:23 liten
-rwxr-xr-x 1 ngift staff 651 Mar 4 22:23 mypython
$ bin/mypython

>>> import liten
```

Наконец, т. к. сценарий «liten» был создан с использованием точки входа, которые обсуждались ранее в этой главе, то при установке пакета формата egg помимо модуля автоматически был установлен консольный сценарий в локальный каталог *bin* в окружении Buildout. Если попробовать запустить этот сценарий, будет получен вывод, как показано ниже:

```
$ bin/liten
Usage: liten [starting directory] [options]
```

```
A command-line tool for detecting duplicates using md5 checksums.

Options:
--version            show program's version number and exit
-h, --help           show this help message and exit
-c, --config         Path to read in config file
-s SIZE, --size=SIZE File Size Example: 10bytes, 10KB, 10MB,10GB,10TB, or
plain number defaults to MB (1 = 1MB)
-q, --quiet          Suppresses all STDOUT.
-r REPORT, --report=REPORT
Path to store duplication report. Default CWD
-t, --test           Runs doctest.
$ pwd
/Users/ngift/src/buildout_demo
```

Это очень простой и яркий пример, демонстрирующий, как можно использовать Buildout для создания изолированной среды и автоматически развертывать все необходимые зависимости проекта и самой среды. Тем не менее, чтобы продемонстрировать истинную мощь Buildout, нам необходимо рассмотреть еще один аспект этого инструмента. Buildout обладает полным «контролем» над каталогом, в котором он выполняется, и при каждом запуске он читает файл *buildout.cfg* в поисках инструкций. Это означает, что если удалить пакет egg из списка, это приведет к удалению инструмента командной строки и библиотеки, как показано в примере 9.6.

Пример 9.6. Удаление записей из конфигурационного файла Buildout

```
[buildout]
parts =
```

Ниже приводится результат повторного запуска Buildout после удаления пакетов и интерпретатора из списка. Обратите внимание, что у Buildout имеется множество параметров командной строки и в данном случае мы указали ключ `-N`, который всего лишь модифицирует измененные файлы. Обычно, на каждом перезапуске Buildout перестраивает полностью свою среду.

```
$ bin/buildout -N
Uninstalling mypython.
```

Если теперь заглянуть в каталог *bin*, можно будет убедиться, что интерпретатор и инструмент командной строки исчезли. Единственное, что осталось в нем, это сам инструмент командной строки Buildout:

```
$ ls -l bin/
total 8
-rwxr-xr-x 1 ngift staff 362 Mar 4 22:17 buildout
```

Однако, если заглянуть в каталог *eggs*, можно увидеть, что пакет установлен, но неактивен. Мы не сможем запустить его, так как интерпретатор отсутствует:

```
$ ls -l eggs
total 640
drwxr-xr-x 7 ngift staff    238 Mar 4 22:54 liten-0.1.3-py2.5.egg
-rw-r--r-- 1 ngift staff 324858 Feb 16 23:47 setuptools-0.6c8-py2.5.egg
drwxr-xr-x 5 ngift staff    170 Mar 4 22:17 zc.buildout-1.0.0-py2.5.egg
drwxr-xr-x 4 ngift staff    136 Mar 4 22:23 zc.recipe.egg-1.0.0-py2.5.egg
```

Разработка с использованием Buildout

Теперь, когда мы рассмотрели простой пример создания и удаления среды, управляемой инструментом Buildout, мы можем двинуться дальше и создать среду разработки, управляемую Buildout.

Один из наиболее типичных сценариев, когда используется Buildout, выглядит очень просто. Разработчик может работать над отдельным пакетом, который находится в репозитарии системы управления версиями. Разработчик извлекает свой проект в каталог верхнего уровня *src*. Внутри этого каталога он запускает Buildout, как было описано выше, примерно с таким конфигурационным файлом:

```
[buildout]
develop = .
parts = test

[python]
recipe = zc.recipe.egg
interpreter = python
eggs = ${config:mypkgs}

[scripts]
recipe = zc.recipe.egg:scripts
eggs = ${config:mypkgs}

[test]
recipe = zc.recipe.testrunner
eggs = ${config:mypkgs}
```

virtualenv

Согласно описанию на странице в Python Package Index: «virtualenv – это инструмент создания изолированной среды Python». Основная задача, которую решает virtualenv, состоит в устранении конфликтов между пакетами. Часто один инструмент требует одну версию некоторого пакета, а другой инструмент – другую версию того же пакета. Это может породить ситуацию, когда будет нарушена работоспособность рабочего веб-приложения, потому что кто-то «случайно» изменит содержимое глобального каталога *site-packages*, обновив пакет, чтобы получить возможность пользоваться другим инструментом.

С другой стороны, разработчик может не иметь права на запись в глобальный каталог *site-packages*, и тогда он может с помощью virtualenv

создать виртуальную среду, изолированную от системной среды Python. Инструмент virtualenv предоставляет отличный способ ликвидировать проблемы еще до того, как они появятся, так как он позволяет создать новую среду, частично или полностью изолированную от глобального каталога *site-packages*.

Инструмент virtualenv также может «развертывать» виртуальную среду, позволяя разработчикам наполнять ее только требуемыми пакетами. По своему действию он очень напоминает Buildout, но в отличие от последнего не использует декларативный конфигурационный файл. Следует заметить, что оба инструмента, Buildout и virtualenv, очень широко используют библиотеку setuptools, сопровождением которой в настоящее время занимается Филлип Дж. Эби (Phillip J. Eby).

ПОРТРЕТ ЗНАМЕНИТОСТИ: VIRTUALENV

Ян Бикинг (Ian Bicking)

Ян Бикинг отвечает за такое количество пакетов Python, что ему часто бывает сложно уследить за всем. Им был написан пакет Webob, являющийся частью Google App Engine, Paste, virtualenv, SQLObject и многие другие пакеты. Вы можете посетить его блог по адресу: *http://blog.ianbicking.org/*.

Так как же пользоваться инструментом virtualenv? Самый простой способ – установить его с помощью `easy_install`:

```
sudo easy_install virtualenv
```

Если вы планируете использовать virtualenv только с одной версией Python, такой подход вполне оправдывает себя. Если у вас установлено несколько версий Python, например, Python 2.4, Python 2.5, Python 2.6 и, возможно, Python 3000, и при этом они установлены в один и тот же каталог *bin*, такой как */usr/bin*, тогда лучше будет использовать иной подход, поскольку в один и тот же каталог можно установить только один сценарий virtualenv.

Один из способов получить несколько сценариев virtualenv, работающих с разными версиями Python, состоит в том, чтобы просто загрузить последнюю версию virtualenv и создать псевдонимы для каждой версии Python. Ниже описывается, как это сделать:

1. curl *http://svn.colorstudy.com/virtualenv/trunk/virtualenv.py* > *virtualenv.py*
2. sudo cp *virtualenv.py /usr/local/bin/virtualenv.py*

3. Создать два псевдонима в командной оболочке Bash или zsh:

```
alias virtualenv-py24="/usr/bin/python2.4 /usr/local/bin/virtualenv.py"
alias virtualenv-py25="/usr/bin/python2.5 /usr/local/bin/virtualenv.py"
alias virtualenv-py26="/usr/bin/python2.6 /usr/local/bin/virtualenv.py"
```

Создав среду с несколькими сценариями, можно двинуться дальше и создать несколько контейнеров virtualenv, по одному для каждой версии Python, которые нам потребуются. Ниже показано, как это делается.

Создание виртуальной среды Python 2.4:

```
$ virtualenv-py24 /tmp/sandbox/py24ENV
New python executable in /tmp/sandbox/py24ENV/bin/python
Installing setuptools.................done.
$ /tmp/sandbox/py24ENV/bin/python
Python 2.4.4 (#1, Dec 24 2007, 15:02:49)
[GCC 4.0.1 (Apple Inc. build 5465)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
$ ls /tmp/sandbox/py24ENV/
bin/ lib/
$ ls /tmp/sandbox/py24ENV/bin/
activate     easy_install*     easy_install-2.4*  python*     python2.4@
```

Создание виртуальной среды Python 2.5:

```
$ virtualenv-py25 /tmp/sandbox/py25ENV
New python executable in /tmp/sandbox/py25ENV/bin/python
Installing setuptools.........................done.
$ /tmp/sandbox/py25ENV/bin/python
Python 2.5.1 (r251:54863, Jan 17 2008, 19:35:17)
[GCC 4.0.1 (Apple Inc. build 5465)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
$ ls /tmp/sandbox/py25ENV/
bin/ lib/
$ ls /tmp/sandbox/py25ENV/bin/
activate     easy_install*    easy_install-2.5*  python*     python2.5@
```

Если внимательно взглянуть на вывод команд, можно заметить, что virtualenv создал каталоги *bin* и *lib*. Внутри каталога *bin* находится интерпретатор `python`, который использует каталог *lib* как собственный локальный каталог *site-packages*. Другой заметной особенностью является наличие предустановленного сценария `easy_install`, что позволяет устанавливать пакеты в виртуальную среду.

Наконец, важно отметить, что существует два способа работы с созданной виртуальной средой. Можно запустить виртуальную среду, явно указав полный путь к интерпретатору:

```
$ /src/virtualenv-py24/bin/python2.4
```

Или можно использовать сценарий `activate`, находящийся в каталоге *bin* требуемой виртуальной среды, чтобы активировать эту среду без ввода полного пути. Это дополнительный способ, который вы можете использовать, но он не является обязательным, так как вы всегда можете ввести полный путь к требуемой виртуальной среде. Дуг Хеллманн (Doug Hellmann), один из рецензентов этой книги, написал интересный сценарий, доступный по адресу: *http://www.doughellmann.com/projects/virtualenvwrapper/*. Он использует сценарий `activate` и меню на языке Bash, которое позволяет выбирать, какая среда должна быть запущена.

Создание собственных виртуальных окружений

Версия virtualenv 1.0, которая была текущей на момент написания этой книги, включает поддержку создания сценариев развертывания собственных виртуальных окружений. Добиться этого можно с помощью функции `virtualenv.create_bootstrap_script(text)`. Эта функция создает сценарий развертывания, который по своему действию напоминает virtualenv, но обладает расширенными возможностями анализа параметров с помощью функций, определяемых пользователем, `extend_parser()` и `adjust_options()`, и позволяет выполнять действия после установки с помощью функции `after_install()`.

Давайте посмотрим, насколько просто создать собственный сценарий развертывания, который установит virualenv и заданный набор пакетов в новую среду. Возьмем опять в качестве примера пакет liten. Мы можем с помощью virtualenv создать совершенно новую виртуальную среду и установить в нее пакет liten. В примере 9.7 показано, как создается сценария развертывания собственной виртуальной среды, который устанавливает пакет liten.

Пример 9.7. Пример сценария, развертывающего новую виртуальную среду

```
import virtualenv, textwrap
output = virtualenv.create_bootstrap_script(textwrap.dedent("""
import os, subprocess
def after_install(options, home_dir):
    etc = join(home_dir, 'etc')
    if not os.path.exists(etc):
        os.makedirs(etc)
    subprocess.call([join(home_dir, 'bin', 'easy_install'),
                     'liten'])
"""))
f = open('liten-bootstrap.py', 'w').write(output)
```

Этот сценарий является измененной версией примера из документации к virtualenv и здесь особое внимание следует обратить на последние две строки:

```
subprocess.call([join(home_dir, 'bin', 'easy_install'),
                 'liten'])
```

```
"""))
f = open('liten-bootstrap.py', 'w').write(output)
```

В двух словах, эти строки предписывают функции `after_install()` выполнить запись в новый файл с именем *liten-bootstrap.py*, расположенный в текущем рабочем каталоге, и затем с помощью `easy_install` установить модуль liten. Важно отметить, что этот фрагмент программного кода создаст файл *bootstrap.py*, который затем будет использоваться при запуске. После запуска этого сценария мы получим файл *liten-bootstrap.py*, который потом можно передать разработчику или конечному пользователю.

Если запустить сценарий `liten-bootstrap.py` без параметров, от него будет получен следующий вывод:

```
$ python liten-bootstrap.py
You must provide a DEST_DIR
Usage: liten-bootstrap.py [OPTIONS] DEST_DIR

Options:
--version          show program's version number and exit
-h, --help         show this help message and exit
-v, --verbose      Increase verbosity
-q, --quiet        Decrease verbosity
--clear            Clear out the non-root install and start from scratch
--no-site-packages Don't give access to the global site-packages dir to the
                   virtual environment
```

Если запустить этот сценарий, указав ему каталог назначения, будет получен следующий вывод:

```
$ python liten-bootstrap.py --no-site-packages /tmp/liten-ENV
New python executable in /tmp/liten-ENV/bin/python
Installing setuptools.........................done.
Searching for liten
Best match: liten 0.1.3
Processing liten-0.1.3-py2.5.egg
Adding liten 0.1.3 to easy-install.pth file
Installing liten script to /tmp/liten-ENV/bin

Using /Library/Python/2.5/site-packages/liten-0.1.3-py2.5.egg
Processing dependencies for liten
Finished processing dependencies for liten
```

Наш интеллектуальный сценарий автоматически создал среду с нашим модулем. Если теперь запустить инструмент liten с полным путем к виртуальной среде, мы получим следующее:

```
$ /tmp/liten-ENV/bin/liten
Usage: liten [starting directory] [options]

A command-line tool for detecting duplicates using md5 checksums.

Options:
```

```
--version             show program's version number and exit
-h, --help            show this help message and exit
-c, --config          Path to read in config file
-s SIZE, --size=SIZE File Size Example: 10bytes, 10KB, 10MB,10GB,10TB, or
plain number defaults to MB (1 = 1MB)
-q, --quiet           Suppresses all STDOUT.
-r REPORT, --report=REPORT
Path to store duplication report. Default CWD
-t, --test            Runs doctest.
```

Этот прием стоит того, чтобы знать о нем, так как он позволяет создавать полностью изолированную и развернутую виртуальную среду.

Мы надеемся, что в этом разделе нам удалось показать одно из основных преимуществ virutalenv – его простоту в использовании и изучении. Больше, чем что бы то ни было, virtualenv почитает священное правило KISS (keep its syntax simple – сохраняй синтаксис как можно проще), и одной этой причины уже достаточно, чтобы подумать об использовании этого инструмента для управления изолированными средами разработки. Если у вас имеются дополнительные вопросы, касающиеся этого инструмента, обязательно посетите почтовую рассылку virtualenv по адресу *http://groups.google.com/group/python-virtualenv/*.

Менеджер пакетов EPM

Менеджер пакетов EPM создает «родные» пакеты для каждой операционной системы, поэтому он должен присутствовать в любой системе, где производится «сборка» пакетов. Благодаря невероятным успехам технологий виртуализации за последние несколько лет не составляет никакого труда установить и настроить несколько виртуальных машин. Я создал маленький кластер виртуальных машин (с минимальным потреблением памяти), которые загружаются в режиме, эквивалентном уровню 3 в Red Hat, – чтобы проверить примеры программного кода, которые приводятся в этой книге.

Впервые возможности EPM продемонстрировал мне мой коллега, который одновременно является одним из разработчиков EPM. Я тогда искал инструмент, который позволил бы мне создавать пакеты программного обеспечения, которое я разрабатывал, в зависимости от типа операционной системы, и он назвал EPM. После прочтения некоторой документации на сайте *http://www.epmhome.org/epm-book.html* я был приятно удивлен, насколько простым и безболезненным оказался процесс создания пакетов. В этом разделе мы пройдем все этапы создания пакета программного обеспечения, готового к установке на самых разных платформах: Ubuntu, OS X, Red Hat, Solaris и FreeBSD. Эти шаги легко могут быть применены к другим системам, поддерживающим EPM, таким как AIX или HP-UX.

Прежде чем перейти к изучению, приведу некоторые начальные сведения о EPM. Согласно официальной документации, этот менеджер пакетов изначально предусматривал сборку дистрибутивов программного обеспечения в двоичном формате, на основе общей спецификации формата. Благодаря такой постановке задачи одни и те же файлы дистрибутивов могли использоваться в любых операционных системах и для всех форматов.

Требования и установка менеджера пакетов EPM

Для установки EPM требуется только командная оболочка типа Bourne shell, компилятор языка C и программы make и gzip. Все эти утилиты легко получить практически в любой UNIX-подобной системе, если они уже не установлены. После того как исходные тексты EPM будут загружены, необходимо выполнить следующую последовательность команд:

```
./configure
make
make install
```

Создание дистрибутива Hello World с инструментом командной строки

Прежде чем приступать к созданию пакетов для любой UNIX-подобной системы, необходимо иметь что-либо, что требуется упаковать. В духе сложившихся традиций мы сначала создадим простой инструмент командной строки с именем `hello_epm.py`, как показано в примере 9.8.

Пример 9.8. Инструмент командной строки Hello EPM

```
#!/usr/bin/env python

import optparse

def main():
    p = optparse.OptionParser()
    p.add_option('--os', '-o', default="*NIX")
    options, arguments = p.parse_args()
    print 'Hello EPM, I like to make packages on %s' % options.os

if __name__ == '__main__':
    main()
```

Если запустить этот сценарий, будет получен следующий вывод:

```
$ python hello_epm.py
Hello EPM, I like to make packages on *NIX

$ python hello_epm.py --os RedHat
Hello EPM, I like to make packages on RedHat
```

Создание платформозависимых пакетов с помощью EPM

«Основы» использования настолько просты, что может вызвать у вас недоумение, почему раньше вы никогда не пользовались EPM для упаковки кросс-платформенного программного обеспечения. EPM читает файл(ы) с «перечнем», описывающим ваш пакет с программным обеспечением. Комментарии в этом «перечне» начинаются с символа #, директивы – с символа %, переменные – с символа $ и, наконец, строки с именами файлов, каталогов, сценариев инициализации и символических ссылок начинаются с алфавитного символа.

С помощью EPM можно создавать как универсальные кросс-платформенные сценарии установки, так и платформозависимые пакеты. Мы сосредоточимся на создании платформозависимых файлов пакетов. Следующий шаг на пути к созданию платформозависимого пакета заключается в создании манифеста, или «перечня», описывающего пакет. В примере 9.9 приводится шаблон манифеста, использовавшийся нами для создания пакета с нашим инструментом командной строки `hello_epm`. Вообще, этот шаблон является настолько универсальным, что вы можете использовать его с незначительными изменениями для создания своих собственных инструментов.

Пример 9.9. Шаблон «перечня» для EPM

```
#Файл перечня для EPM может использоваться для создания пакетов под
#любую из следующих платформ
#epm -f format foo bar.list ENTER
#Параметр format может быть одним из следующих ключевых слов:

#aix - пакеты с программным обеспечением для AIX.
#bsd - пакеты с программным обеспечением для FreeBSD, NetBSD или OpenBSD.
#depot или swinstall - пакеты с программным обеспечением для HP-UX.
#dpkg - пакеты с программным обеспечением для Debian.
#inst или tardist - пакеты с программным обеспечением для IRIX.
#native - "родные" для текущей платформы пакеты (RPM, INST, DEPOT, PKG, ...).
#osx - пакеты с программным обеспечением для MacOS X.
#pkg - пакеты с программным обеспечением для Solaris.
#portable - переносимые пакеты с программным обеспечением (по умолчанию).
#rpm - пакеты с программным обеспечением для Red Hat.
#setld - пакеты с программным обеспечением для Tru64 (setld).
#slackware - пакеты с программным обеспечением для Slackware.

# Раздел с информацией о продукте

%product hello_epm
%copyright 2008 Py4SA
%vendor O'Reilly
%license COPYING
%readme README
%description Command Line Hello World Tool
%version 0.1
```

```
# Переменные для сценария автоматической конфигурации

$prefix=/usr
$exec_prefix=/usr
$bindir=${exec_prefix}/bin
$datadir=/usr/share
$docdir=${datadir}/doc/
$libdir=/usr/lib
$mandir=/usr/share/man
$srcdir=.

# Выполняемые файлы

%system all
f 0555 root sys ${bindir}/hello_epm hello_epm.py

# Документация

%subpackage documentation
f 0444 root sys ${docdir}/README $srcdir/README
f 0444 root sys ${docdir}/COPYING $srcdir/COPYING
f 0444 root sys ${docdir}/hello_epm.html $srcdir/doc/hello_epm.html

# Файлы страниц справочного руководства (man)

%subpackage man
%description Man pages for hello_epm
f 0444 root sys ${mandir}/man1/hello_epm.1 $srcdir/doc/hello_epm.man
```

Если заглянуть внутрь файла, который мы назвали *hello_epm.list*, можно заметить, что мы определили переменную `$srcdir`, значение которой соответствует текущему рабочему каталогу. Чтобы создать пакет для любой из возможных платформ, нам необходимо создать в текущем рабочем каталоге следующие файлы и каталоги: *README*, *COPYING*, *doc/hello_epm.html* и *doc/hello_epm.man*. В этом же каталоге должен находиться и наш сценарий `hello_epm.py`.

При желании мы можем «обмануть» EPM, просто поместив пустые файлы в каталог, который предполагается упаковать, выполнив следующие команды:

```
$ pwd
/tmp/release/hello_epm
$ touch README
$ touch COPYING
$ mkdir doc
$ touch doc/hello_epm.html
$ touch doc/hello_epm.man
```

Если теперь заглянуть в наш каталог, можно увидеть следующее его содержимое:

```
$ ls -lR
total 16
-rw-r--r--  1 ngift wheel    0 Mar 10 04:45 COPYING
```

```
-rw-r--r--  1 ngift wheel    0 Mar 10 04:45 README
drwxr-xr-x  4 ngift wheel  136 Mar 10 04:45 doc
-rw-r--r--  1 ngift wheel 1495 Mar 10 04:44 hello_epm.list
-rw-r--r--@ 1 ngift wheel  278 Mar 10 04:10 hello_epm.py

./doc:
total 0
-rw-r--r--  1 ngift wheel    0 Mar 10 04:45 hello_epm.html
-rw-r--r--  1 ngift wheel    0 Mar 10 04:45 hello_epm.man
```

Создание пакета

Теперь у нас имеется каталог с файлом «перечня», содержащий директивы, которые могут быть выполнены на любой платформе, где имеется поддержка EPM. Теперь все, что осталось сделать, это запустить команду `epm -f`, добавив к ней название платформы и имя файла перечня. В примере 9.10 показано, как это выглядит в OS X.

Пример 9.10. Создание «родного» для OS X инсталлятора с помощью EPM

```
$ epm -f osx hello_epm hello_epm.list
epm: Product names should only contain letters and numbers!
^C
$ epm -f osx helloEPM hello_epm.list
$ ll
total 16
-rw-r--r--  1 ngift wheel    0 Mar 10 04:45 COPYING
-rw-r--r--  1 ngift wheel    0 Mar 10 04:45 README
drwxr-xr-x  4 ngift wheel  136 Mar 10 04:45 doc
-rw-r--r--  1 ngift wheel 1495 Mar 10 04:44 hello_epm.list
-rw-r--r--@ 1 ngift wheel  278 Mar 10 04:10 hello_epm.py
drwxrwxrwx  6 ngift staff  204 Mar 10 04:52 macosx-10.5-intel
```

Обратите внимание на предупреждение, которое было получено при попытке использовать символ подчеркивания в имени пакета. По этой причине мы дали пакету другое название и повторно запустили команду. В результате был создан каталог *macosx-10.5-intel* со следующим содержимым:

```
$ ls -la macosx-10.5-intel
total 56
drwxrwxrwx  4 ngift staff   136 Mar 10 04:54 .
drwxr-xr-x  8 ngift wheel   272 Mar 10 04:54 ..
-rw-r--r--@ 1 ngift staff 23329 Mar 10 04:54 helloEPM-0.1-macosx-10.5-
intel.dmg
drwxr-xr-x  3 ngift wheel   102 Mar 10 04:54 helloEPM.mpkg
```

Это очень удобно, так как был создан файл архива *.dmg*, который является «родным» форматом для OS X, содержащий наш инсталлятор и инсталлятор, «родной» для OS X.

Если теперь запустить установку, можно заметить, что OS X установит пустые файлы со страницей справочного руководства и с документа-

цией и выведет содержимое пустого файла с лицензионным соглашением. В конечном итоге наш инструмент будет помещен точно туда, куда было указано, и ему будет присвоено заданное нами имя:

```
$ which hello_epm
/usr/bin/hello_epm
$ hello_epm
Hello EPM, I like to make packages on *NIX
$ hello_epm -h
Usage: hello_epm [options]

Options:
-h, --help show this help message and exit
-o OS, --os=OS
$
```

Вывод: EPM действительно прост в использовании

Если с помощью команды `scp -r` скопировать каталог `/tmp/release/hello_epm` в Red Hat, Ubuntu или Solaris, мы сможем выполнить одну и ту же команду создания пакета, за исключением названия платформы, и она «просто будет работать». В главе 8 мы узнали, как создать «ферму» для сборки, чтобы вы могли моментально создавать кросс-платформенные пакеты. Обратите внимание, что все представленные исходные тексты примеров наряду с созданным пакетом, доступны для загрузки. Теперь у вас есть все необходимые знания, чтобы за несколько минут, немного изменив пример, начать создавать свои собственные кросс-платформенные пакеты.

Менеджер пакетов EPM в состоянии предложить еще целый ряд дополнительных особенностей, но они выходят далеко за рамки этой книги. Если вам интересно узнать о том, как создавать пакеты с учетом зависимостей, как запускать пред- и постустановочные сценарии и так далее, то вам следует обратиться непосредственно к официальной документации EPM, где описываются все эти случаи и многое другое.

10

Процессы и многозадачность

Введение

Обращение с процессами для системного администратора UNIX/Linux – это реалии жизни. Вы должны знать о сценариях запуска системных служб, уровнях запуска, демонах, о заданиях планировщика cron, о долгоживущих процессах, о многозадачности и о массе других проблем. К счастью, язык Python делает работу с процессами удивительно простым делом. Начиная с версии Python 2.4, появился универсальный модуль subprocess, позволяющий порождать новые процессы и обмениваться информацией с ними через устройства стандартного ввода, стандартного вывода и стандартного вывода сообщений об ошибках. Обмен информацией с процессами – это лишь один из аспектов работы с ними; не менее важно понимать, как развертывать и управлять процессами, работающими продолжительное время.

Модуль subprocess

В версии Python 2.4 появился новый модуль subprocess, занявший место нескольких старых модулей: `os.system`, `os.spawn`, `os.popen` и `popen2`. Модуль subprocess принес революционные изменения в жизнь системных администраторов и разработчиков, которым приходится иметь дело с процессами и постоянно прибегать к командам оболочки. Теперь имеется универсальный модуль для работы с процессами, который, в конечном счете, может использоваться для управления группами процессов.

Модуль subprocess можно считать самым важным для системного администратора модулем, потому что он обеспечивает унифицированный интерфейс к системе. Модуль subprocess отвечает в языке Python за выполнение следующих действий: порождение новых процессов,

соединение с потоками стандартного ввода, стандартного вывода, стандартного вывода сообщений об ошибках и получение кодов возврата от этих процессов.

Чтобы подогреть ваш интерес, будем следовать принципу KISS (Keep Its Syntax Simple – сохраняй синтаксис как можно проще) и с помощью модуля subprocess выполним простейший вызов системной утилиты, как показано в примере 10.1.

Пример 10.1. Простейший пример использования модуля subprocess

```
In [4]: subprocess.call('df -k', shell=True)
Filesystem    1024-blocks      Used  Available  Capacity  Mounted on
/dev/disk0s2     97349872  80043824   17050048     83%    /
devfs                106       106          0    100%    /dev
fdesc                  1         1          0    100%    /dev
map -hosts             0         0          0    100%    /net
map auto_home          0         0          0    100%    /home
Out[4]: 0
```

Используя тот же простой синтаксис, можно использовать переменные окружения. В примере 10.2 показано, как можно получить объем дискового пространства, занимаемого домашним каталогом.

Пример 10.2. Объем используемого дискового пространства

```
In [7]: subprocess.call('du -hs $HOME', shell=True)
28G    /Users/ngift
Out[7]: 0
```

Следует отметить один интересный прием, позволяющий при использовании модуля subprocess подавить вывод в поток стандартного вывода. Многие интересуются только возможностью запускать команды системы, но никак не беспокоятся о стандартном выводе. Часто в таких случаях бывает необходимо подавить стандартный вывод вызова `subprocess.call()`. К счастью, сделать это очень просто, как показано в примере 10.3.

Пример 10.3. Подавление стандартного вывода вызова subprocess.call( )

```
In [3]: import subprocess

In [4]: ret = subprocess.call("ping -c 1 10.0.1.1",
                              shell=True,
                              stdout=open('/dev/null', 'w'),
                              stderr=subprocess.STDOUT)
```

По поводу этих двух примеров и вызова `subprocess.call()` имеется несколько общих примечаний. Обычно при использовании функции `subprocess.call()` необходимо просто запустить команду, а вывод от нее сохранять не требуется. Если же необходимо захватить вывод команды, то следует использовать функцию `subprocess.Popen()`. Между функциями `subprocess.call()` и `subprocess.Popen()` существует еще одно су-

щественное отличие. Функция `subprocess.call()` блокирует выполнение сценария до получения ответа, в то время как функция `subprocess.Popen()` – нет.

Использование кодов возврата с помощью модуля subprocess

Интересно заметить, что при использовании `subprocess.call()` можно получать коды возврата, чтобы определить, насколько успешно была выполнена команда. Если у вас есть опыт программирования на языке C или Bash, вы должны быть близко знакомы с кодами возврата. Часто взаимозаменяемые фразы «код выхода» или «код возврата» используются для обозначения кода состояния системного процесса.

Каждый процесс возвращает код возврата при выходе, и значение кода возврата может использоваться, чтобы определить, какие действия должна предпринять программа. Вообще, если программа возвращает значение, отличное от нуля, это свидетельствует об ошибке. Самое очевидное для разработчика использование кода возврата – определить, какие действия выполнять, если процесс вернул значение кода, отличное от нуля, свидетельствующее об ошибке. Но существует множество более интересных, хотя и не таких очевидных способов использования кодов возврата. Существуют специальные значения кодов возврата, которые свидетельствуют о том, что искомый объект не найден, что файл не является исполняемой программой или программа была завершена комбинацией клавиш Ctrl-C. В этом разделе мы будем использовать все эти коды возвратов в программах на языке Python.

В следующем списке приводятся наиболее распространенные коды возврата с их назначением:

0 Успешное завершение

1 Общая ошибка

2 Неправильное использование встроенных команд оболочки

126 Вызываемая команда не может быть выполнена

127 Команда не найдена

128 Неверный аргумент команды `exit`

128+n Фатальная ошибка по сигналу «n»

130 Сценарий был завершен нажатием комбинации клавиш Ctrl-C

255 Указан код завершения за пределами допустимого диапазона

Наиболее практичный пример, где эти сведения могли бы применяться, – это использование кодов 0 и 1, которые просто свидетельствуют об успешном или неудачном завершении только что выполненной команды. Рассмотрим несколько простых примеров использования кодов, возвращаемых функцией `subprocess.call()`. Взгляните на пример 10.4.

Пример 10.4. Код, возвращаемый функцией subprocess.call( ) в случае неудачи

```
In [16]: subprocess.call("ls /foo", shell=True)
ls: /foo: No such file or directory
Out[16]: 1
```

Поскольку этот каталог отсутствует, мы получили код возврата 1, свидетельствующий об ошибке. Мы можем записывать код возврата в переменную и затем использовать его в условных инструкциях, как показано в примере 10.5.

Пример 10.5. Условные инструкции, проверяющие код возврата, получаемый от функции subprocess.call( )

```
In [25]: ret = subprocess.call("ls /foo", shell=True)
ls: /foo: No such file or directory
In [26]: if ret == 0:
   ....:     print "success"
   ....: else:
   ....:     print "failure"
   ....:
   ....:
failure
```

Ниже приводится пример получения кода возврата «команда не найдена», который имеет значение 127. Это может быть полезно для создания инструмента, который может пытаться запускать различные похожие команды оболочки на основе информации об их доступности. Например, можно было бы сначала попробовать запустить команду `rsync`, и если будет получен код возврата 127, попытаться выполнить команду `scp -r`, как показано в примере 10.6.

Пример 10.6. Код 127, возвращаемый функцией subprocess.call( )

```
In [28]: subprocess.call("rsync /foo /bar", shell=True)
/bin/sh: rsync: command not found
Out[28]: 127
```

Возьмем предыдущий пример и сделаем его менее абстрактным. Часто при создании кросс-платформенного программного кода, который должен работать в различных UNIX-подобных системах, вы можете столкнуться с ситуацией, когда для достижения определенного результата необходимо использовать различные системные программы в зависимости от того, в какой операционной системе выполняется сценарий. Каждая из систем HP-UX, AIX, Solars, FreeBSD и Red Hat может иметь разные утилиты, которые делают то, что вам требуется. Сценарий мог бы попытаться выполнить с помощью модуля subprocess сначала одну команду, а получив код возврата 127, попытаться выполнить другую команду, и так далее.

К сожалению, значения кодов возврата могут изменяться от системы к системе, поэтому, если вы пишете кросс-платформенный сценарий, возможно, желательнее будет анализировать лишь нулевое и ненуле-

вое значение кода выхода. Ради примера, ниже показан код возврата, который был получен в Solaris 10 при выполнении той же команды, что раньше выполнялась в Red Hat Enterprise Linux 5:

```
ash-3.00# python
Python 2.4.4 (#1, Jan 9 2007, 23:31:33) [C] on sunos5
Type "help", "copyright", "credits" or "license" for more information.
>>> import subprocess
>>> subprocess.call("rsync", shell=True)
    /bin/sh: rsync: not found
    1
```

Мы по-прежнему можем использовать определенные коды возврата, предварительно определив тип операционной системы. После определения типа системы можно было бы проверить наличие определенной команды. Если вы предполагаете писать такой программный код, тогда для вас будет совсем нелишним познакомиться поближе с модулем platform. О работе с этим модулем подробно рассказывалось в главе 8, поэтому вы можете обращаться к ней за дополнительной информацией.

Взгляните на пример 10.7, в котором модуль platform используется в интерактивном режиме в оболочке IPython, чтобы определить, какую команду передать функции `subprocess.call()`.

Пример 10.7. Использование модулей platform и subprocess, чтобы убедиться, что команда выполняется в Solaris 10

```
In [1]: import platform
In [2]: import subprocess
In [3]: platform?

        Namespace: Interactive
File:              /usr/lib/python2.4/platform.py
Docstring:
This module tries to retrieve as much platform-identifying data as
possible. It makes this information available via function APIs.
(Этот модуль пытается получить максимально возможный объем информации,
идентифицирующей операционную систему, и обеспечивает доступ к этой
информации через функции API.)

If called from the command line, it prints the platform
information concatenated as single string to stdout. The output
format is useable as part of a filename.
(При вызове из командной строки выводит на устройство stdout информацию
о платформе в виде единой строки. Строка имеет такой формат, что может
использоваться как имя файла.)

In [4]: if platform.system() == 'SunOS':
  ....:     print "yes"
  ....:
yes

In [5]: if platform.release() == '5.10':
  ....:     print "yes"
```

```
  ....:
yes

In [6]: if platform.system() == 'SunOS':
   ...:     ret = subprocess.call('cp /tmp/foo.txt /tmp/bar.txt', shell=True)
   ...:     if ret == 0:
   ...:         print "Success, the copy was made on %s %s " %
(platform.system(), platform.release())
   ...:
Success, the copy was made on SunOS 5.10
```

Как видите, модуль platform в соединении с функцией `subprocess.call()` может оказаться эффективным средством в создании кросс-платформенного программного кода. За подробной информацией об использовании модуля platform при создании кросс-платформенного программного кода для UNIX-подобных систем обращайтесь к главе 8. Взгляните на пример 10.8.

Пример 10.8. Захват вывода от команды средствами модуля subprocess

```
In [1]: import subprocess

In [2]: p = subprocess.Popen("df -h", shell=True, stdout=subprocess.PIPE)

In [3]: out = p.stdout.readlines()

In [4]: for line in out:
   ...:     print line.strip()
   ...:
   ...:
Filesystem     Size    Used    Avail  Capacity   Mounted on
/dev/disk0s2   93Gi    78Gi     15Gi     85%     /
devfs         107Ki   107Ki      0Bi    100%     /dev
fdesc         1.0Ki   1.0Ki      0Bi    100%     /dev
map -hosts      0Bi     0Bi      0Bi    100%     /net
map auto_home   0Bi     0Bi      0Bi    100%     /home
```

Обратите внимание, что метод `readlines()` возвращает список, в котором строки завершаются символом новой строки. Для их удаления мы использовали вызов метода `line.strip()`. Кроме того, модуль subprocess поддерживает возможность взаимодействия с устройствами стандартного ввода и стандартного вывода для создания каналов (или цепочек команд). Ниже приводится простой пример взаимодействия с устройством стандартного вывода процесса. На языке Python можно реализовать такую интересную вещь, как «фабрику» цепочек команд, которая выглядела бы на на языке Bash устрашающе. Всего несколько строк простого программного кода, и мы можем выполнять и выводить результаты последовательности команд, число которых определяется числом аргументов, как показано в примере 10.9.

Пример 10.9. «Фабрика» команд с использованием модуля subprocess

```
def multi(*args):
    for cmd in args:
```

```
        p = subprocess.Popen(cmd, shell=True, stdout = subprocess.PIPE)
        out = p.stdout.read()
        print out
```

Ниже приводится пример этой простой функции в действии:

```
In [28]: multi("df -h", "ls -l /tmp", "tail /var/log/system.log")
Filesystem      Size   Used  Avail Capacity  Mounted on
/dev/disk0s2    93Gi   80Gi   13Gi    87%    /
devfs          107Ki  107Ki    0Bi   100%    /dev
fdesc          1.0Ki  1.0Ki    0Bi   100%    /dev
map -hosts       0Bi    0Bi    0Bi   100%    /net
map auto_home    0Bi    0Bi    0Bi   100%    /home

lrwxr-xr-x@ 1 root admin 11 Nov 24 23:37 /tmp -> private/tmp

Feb 21 07:18:50 dhcp126 /usr/sbin/ocspd[65145]: starting
Feb 21 07:19:09 dhcp126 login[65151]: USER_PROCESS: 65151 ttys000
Feb 21 07:41:05 dhcp126 login[65197]: USER_PROCESS: 65197 ttys001
Feb 21 07:44:24 dhcp126 login[65229]: USER_PROCESS: 65229 ttys002
```

Благодаря мощи языка Python и синтаксической конструкции `*args` мы можем запускать последовательности из произвольного числа команд, используя нашу функцию в качестве фабрики. Каждая команда извлекается из начала списка методом `args.pop(0)`.[1] Если бы мы использовали вызов метода без аргумента `args.pop()`, команды извлекались бы в обратном порядке. Поскольку такой способ извлечения команд может приводить к путанице, мы переписали функцию, реализовав ее на основе простого цикла `for`:[2]

```
def multi(*args):
    for cmd in args:
        p = subprocess.Popen(cmd, shell=True, stdout = subprocess.PIPE)
        out = p.stdout.read()
        print out
```

---

[1] Метод `pop()` в приведенном примере не используется, кроме того, `args` – это кортеж, а у кортежей нет метода `pop()`. Вероятно, раньше, в черновом варианте книги, этот пример был реализован иначе, с преобразованием кортежа аргументов в список аргументов. Такая функция могла бы выглядеть примерно так:

```
def multi(*args):
    cmd = list(args)
    while len(cmd) > 0:
        p = subprocess.Popen(cmd.pop(0), shell=True,
                                         stdout = subprocess.PIPE)
        out = p.stdout.read()
        print out
```

Прим. перев.

[2] Этот пример полностью повторяет предыдущий... См. сноску 1. – *Прим. перев.*

Системным администраторам часто приходится запускать последовательности команд, поэтому определенно имеет смысл создать модуль, который упростил бы эту возможность. Давайте посмотрим, как можно было бы реализовать такой модуль с применением механизма наследования. Исходный текст модуля приводится в примере 10.10.

Пример 10.10. Модуль-обертка вокруг модуля subprocess

```
#!/usr/bin/env python
from subprocess import call
import time
import sys

"""
subtube - это модуль, упрощающий и автоматизирующий некоторые аспекты
применения subprocess
"""

class BaseArgs(object):
    """Основной класс, выполняющий разбор именованных аргументов"""

    def __init__(self, *args, **kwargs):
        self.args = args
        self.kwargs = kwargs
        if self.kwargs.has_key("delay"):
            self.delay = self.kwargs["delay"]
        else:
            self.delay = 0
        if self.kwargs.has_key("verbose"):
            self.verbose = self.kwargs["verbose"]
        else:
            self.verbose = False

    def run (self):
        """Вы должны реализовать метод run"""
        raise NotImplementedError

class Runner(BaseArgs):
    """
    Упрощает вызов subprocess.call и запускает последовательность команд.

    Конструктор класса Runner принимает N позиционных аргументов
    и следующие необязательные аргументы:

    [необязательные именованные аргументы]
    delay=1, задержка в секундах
    verbose=True, подробный отчет о выполняемых действиях

    Порядок использования:

    cmd = Runner("ls -l", "df -h", verbose=True, delay=3)
    cmd.run()
    """

    def run(self):
        for cmd in self.args:
```

```
        if self.verbose:
            print "Running %s with delay=%s" % (cmd, self.delay)
        time.sleep(self.delay)
        call(cmd, shell=True)
```

А теперь посмотрим, как пользоваться нашим новым модулем:

```
In [8]: from subtube import Runner
In [9]: r = Runner("df -h", "du -h /tmp")

In [10]: r.run()
Filesystem      Size   Used  Avail Capacity  Mounted on
/dev/disk0s2    93Gi   80Gi   13Gi    87%    /
devfs          107Ki  107Ki    0Bi   100%    /dev
fdesc          1.0Ki  1.0Ki    0Bi   100%    /dev
map -hosts       0Bi    0Bi    0Bi   100%    /net
map auto_home    0Bi    0Bi    0Bi   100%    /home
4.0K /tmp

In [11]: r = Runner("df -h", "du -h /tmp", verbose=True)

In [12]: r.run()
Running df -h with delay=0
Filesystem      Size   Used  Avail Capacity  Mounted on
/dev/disk0s2    93Gi   80Gi   13Gi    87%    /
devfs          107Ki  107Ki    0Bi   100%    /dev
fdesc          1.0Ki  1.0Ki    0Bi   100%    /dev
map -hosts       0Bi    0Bi    0Bi   100%    /net
map auto_home    0Bi    0Bi    0Bi   100%    /home
Running du -h /tmp with delay=0
4.0K /tmp
```

Если представить, что у нас настроен доступ через ssh ко всем нашим машинам, мы легко могли бы выполнить, например, такое действие:

```
machines = ['homer', 'marge','lisa', 'bart']
for machine in machines:
    r = Runner("ssh " + machine + "df -h", "ssh " + machine + "du -h /tmp")
    r.run()
```

Это топорный пример запуска команд на удаленной машине, но сама идея достойна более пристального внимания, потому что в группе Red Hat Emerging Technology разрабатывается проект, упрощающий управление крупными кластерами компьютеров из сценариев на языке Python. Согласно информации, которая приводится на веб-сайте Func, «Ниже приводится интересный, хотя и искусственный пример выполнения перезагрузки всех систем, где запущен демон httpd. Искусственный, да, но реализовать не составляет труда, благодаря Func». О Func (FUNC) упоминалось в главе 8, где рассматривалась собственная система «управления», способная работать в любой UNIX-подобной системе.

```
results = fc.Client("*").service.status("httpd")
for (host, returns) in results.iteritems():
```

```
    if returns == 0:
        fc.Client(host).reboot.reboot()
```

Модуль subprocess обеспечивает унифицированный интерфейс взаимодействия с системой, с его помощью достаточно легко организовать запись данных в поток стандартного ввода. В примере 10.11 мы запускаем утилиту подсчета числа слов, предлагая ей подсчитать количество символов, и записываем в ее поток стандартного ввода строку символов.

Пример 10.11. Организация связи с потоком стандартного ввода через модуль subprocess

```
In [35]: p = subprocess.Popen("wc -c", shell=True, stdin=subprocess.PIPE)
In [36]: p.communicate("charactersinword")
      16
```

Эквивалентная команда на языке Bash выглядит, как показано ниже:

```
> echo charactersinword | wc -c
```

Попробуем на этот раз сымитировать поведение Bash и перенаправить файл в поток стандартного ввода. Для начала нам необходимо записать что-нибудь в файл; сделаем это с использованием нового синтаксиса Python 2.6. Запомните, что при использовании Python 2.5 необходимо использовать идиому импорта будущих возможностей:

```
In [5]: from __future__ import with_statement
In [6]: with open('temp.txt', 'w') as file:
   ...:     file.write('charactersinword')
```

Теперь можно повторно открыть файл привычным способом и прочитать его содержимое в виде строки, присвоив полученное значение переменной `f`:

```
In [7]: file = open('temp.txt')

In [8]: f = file.read()
```

После этого можно «перенаправить» файл на вход ожидающего процесса:

```
In [9]: p = subprocess.Popen("wc -c", shell=True, stdin=subprocess.PIPE)

In [10]: p.communicate(f)

In [11]: p.communicate(f)
16
```

В командной оболочке Bash эквивалентная последовательность команд выглядит, как показано ниже:

```
% echo charactersinword > temp.txt
% wc -c < temp.txt
16
```

Теперь посмотрим, как реализовать конвейерную обработку с применением нескольких команд, которая часто используется в сценариях на языке командной оболочки. Посмотрим сначала, как выглядит последовательность команд, объединенных в конвейер, на языке Bash, а затем реализуем ту же самую последовательность на языке Python. На практике нам очень часто приходится иметь дело с файлами журналов. В примере 10.12 мы определяем, какая командная оболочка используется суперпользователем root на ноутбуке Macintosh.

Пример 10.12. Объединение команд в цепочку с помощью модуля subprocess

На языке Bash это действие выполняется следующей простой цепочкой команд:

```
[ngift@Macintosh-6][H:10014]> cat /etc/passwd | grep 0:0 | cut -d ':' -f 7
/bin/sh
```

Аналогичная последовательность на языке Python:

```
In [7]: p1 = subprocess.Popen("cat /etc/passwd", shell=True,
    stdout=subprocess.PIPE)
In [8]: p2 = subprocess.Popen("grep 0:0", shell=True, stdin=p1.stdout,
    stdout=subprocess.PIPE)
In [9]: p3 = subprocess.Popen("cut -d ': ' -f 7", shell=True,
stdin=p2.stdout,
   stdout=subprocess.PIPE)

In [10]: print p3.stdout.read()
/bin/sh
```

Тем не менее, хотя мы можем реализовать некоторое действие с помощью модуля subprocess, организовав каналы, но это еще не означает, что только так и следовало действовать. В предыдущем примере мы получили имя командной оболочки пользователя root, объединив в конвейер последовательность команд. Но для выполнения подобных действий в языке Python имеется встроенный модуль, поэтому очень важно знать, когда можно избежать использования модуля subprocess – язык Python может содержать встроенный модуль, который способен выполнить необходимое действие. Многое из того, что можно сделать в командной оболочке, например, создать архив в формате tar или zip, можно реализовать и на языке Python без использования команд системы. Поэтому, когда вы обнаруживаете, что приходится реализовывать очень сложную конвейерную обработку с использованием модуля subprocess, поищите встроенный эквивалент в языке Python. Взгляните на пример 10.13.

Пример 10.13. Использование модуля pwd для работы с базой данных паролей вместо subprocess

```
In [1]: import pwd

In [2]: pwd.getpwnam('root')
Out[2]: ('root', '********', 0, 0, 'System Administrator', '/var/root',
'/bin/sh')
```

```
In [3]: shell = pwd.getpwnam('root')[-1]

In [4]: shell
Out[4]: '/bin/sh'
```

Модуль subprocess позволяет одновременно передавать данные в поток стандартного ввода и принимать данные из потока стандартного вывода процесса, а также получать данные из потока стандартного вывода сообщений об ошибках. Рассмотрим пример, демонстрирующий это.

Обратите внимание, что мы используем команду `ed upper.py` для автоматического переключения в редактор Vim из интерактивной оболочки IPython, когда нам необходимо написать фрагмент программного кода, который может представлять собой блок, аналогичный приведенному в примере 10.14.

Пример 10.14. Передача данных в поток стандартного ввода и прием данных из потока стандартного вывода и из потока стандартного вывода сообщений об ошибках

```
import subprocess

p = subprocess.Popen("tr a-z A-Z", shell=True, stdin=subprocess.PIPE,
                     stdout=subprocess.PIPE)
output, error = p.communicate("translatetoupper")
print output
```

Когда происходит возврат из редактора в оболочку IPython, она автоматически выполняет фрагмент программного кода, и мы получаем следующий результат:

```
done. Executing edited code...
TRANSLATETOUPPER
```

Использование программы Supervisor для управления процессами

Системному администратору часто приходится управлять процессами. Когда веб-разработчики узнают, что их системный администратор является специалистом в языке Python, они очень удивляются, потому что очень немногие веб-платформы на языке Python предлагают элегантные способы управления долгоживущими процессами. Программа Supervisor поможет в ситуациях, когда необходимо организовать управление долгоживущими процессами, и обеспечит их повторный запуск после перезагрузки системы.

В действительности программа Supervisor может значительно больше, чем просто оказывать помощь в развертывании веб-приложений, – у нее есть масса применений общего характера. Supervisor может использоваться как кросс-платформенный контроллер управления процессами и взаимодействия с ними. Supervisor может запускать, останавливать и перезапускать другие программы в UNIX-подобных системах. Кроме

того, Supervisor может выполнять перезапуск «обрушившихся» процессов, что может оказаться очень удобным. Соавтор программы Supervisor, Крис Макдоног (Chris McDonough), сообщил нам, что она может также использоваться для управления «плохими» процессами, то есть процессами, потребляющими, например, слишком много памяти или процессорного времени. Supervisor обеспечивает возможность удаленного управления посредством XML-RPC Interface Extensions Event Notification System.

Основной интерес для большинства администраторов UNIX-подобных систем будут представлять программы `supervisord` – демон, который запускает программы как дочерние процессы, и `supervisorctl` – клиентская программа, позволяющая просматривать файлы журналов и управлять процессами. Кроме того, существует и веб-интерфейс, но, поскольку эта книга о UNIX-подобных системах, двинемся дальше.

К моменту написания этих строк последней была версия программы Supervisor 3.0.x. Последнюю версию руководства к программе всегда можно получить по адресу *http://supervisord.org/manual/current/*. Установка программы Supervisor не вызывает никаких сложностей – ее можно установить с помощью утилиты `easy_install`. Предположим, что мы с помощью virtualenv создали отдельный каталог для изолированной среды Python, в этом случае установить программу Supervisor можно с помощью следующей команды:

```
bin/easy_install supervisor
```

Она установит Supervisor в каталог *bin*. Если воспользоваться утилитой `easy_install` в системной среде Python, то установка будет выполнена в каталог, например, */usr/local/bin* или в каталог по умолчанию для сценариев.

Следующий этап, который следует выполнить перед запуском демона программы Supervisor, заключается в создании простого сценария, который, как в следующем примере, выводит текст, ожидает 3 секунды и завершает свою работу. Такой сценарий, конечно, нельзя назвать долгоживущим процессом, но с его помощью мы продемонстрируем одну из самых сильных сторон программы Supervisor – способность автоматически перезапускать программы, превращая их в некоторое подобие демонов. Теперь можно заполнить файл *supervisord.conf*, используя для этого специальную команду `echo_supervisord_conf`. В этом примере мы просто заполняем файл */etc/supervisord.conf*. Следует отметить, что конфигурационный файл программы Supervisor может находиться в любом месте, потому что демон `supervisord` можно запускать с параметром, указывающим его местоположение.

```
echo_supervisord_conf > /etc/supervisord.conf
```

Выполнив эти подготовительные действия, мы готовы приступить к созданию очень простого примера процесса, который будет завершаться через несколько секунд после запуска. Чтобы обеспечить непрерыв-

ную работу процесса, мы воспользуемся возможностью программы Supervisor перезапускать процессы, как показано в примере 10.15.

Пример 10.15. Простой пример перезапуска процесса с помощью программы Supervisor

```
#!/usr/bin/env python
import time
print "Daemon runs for 3 seconds, then dies"
time.sleep(3)
print "Daemons dies"
```

Как уже упоминалось ранее, чтобы обеспечить запуск дочерних процессов под управлением `supervisord`, нам необходимо отредактировать конфигурационный файл и добавить в него наше приложение. Давайте двинемся дальше и добавим в файл */etc/supervisord.conf* пару строк:

```
[program:daemon]
command=/root/daemon.py ; программа (можно указывать относительные пути
                        ; с учетом переменной PATH и передавать аргументы)
autorestart=true        ; перезапускать при необходимости (по умолчанию:
true)
```

Теперь можно запустить демон `supervisord` и затем с помощью программы `supervisorctl` запускать процессы и следить за ними:

```
[root@localhost]~# supervisord
[root@localhost]~# supervisorctl
daemon                           RUNNING    pid 32243, uptime 0:00:02
supervisor>
```

Здесь мы можем воспользоваться командой `help`, чтобы ознакомиться с доступными параметрами программы `supervisorctl`:

```
supervisor> help

Documented commands (type help topic):
========================================
EOF    exit  maintail   quit   restart  start  stop  version
clear  help  open       reload shutdown status tail
```

Теперь запустим наш процесс, которому в конфигурационном файле мы дали имя daemon, и затем будем следить за его работой, пока он не завершится, после чего он волшебным образом будет перезапущен, почти как Франкенштейн. Процесс живет, умирает и снова оживает.

```
supervisor> stop daemon
daemon: stopped
supervisor> start daemon
daemon: started
```

И в заключение нашей игры мы можем в интерактивном режиме просматривать, что выводится этой программой в поток стандартного вывода:

```
supervisor> tail -f daemon
== Press Ctrl-C to exit ==
    for 3 seconds, then die
    Daemon died
    Daemon runs for 3 seconds, then dies
```

Использование программы screen для управления процессами

Альтернативный подход к управлению процессами заключается в использовании программы GNU screen. Как системному администратору вам необходимо умение работать с программой screen, даже если вы не собираетесь управлять программами из сценариев на языке Python. Одна из основных особенностей программы screen заключается в том, что она позволяет отсоединяться от долгоживущего процесса и вновь возвращаться к нему. Это настолько полезная возможность, что на наш взгляд владение этой программой можно рассматривать как один из основных навыков работы с системой UNIX.

Рассмотрим типичную ситуацию, когда могло бы потребоваться отсоединиться от долгоживущего веб-приложения, такого как trac. Существует несколько способов настройки trac, но самый простой состоит в том, чтобы отсоединиться от отдельного процесса trac с помощью программы screen.

Все, что необходимо для запуска процесса под управлением программы screen, – это поместить команду screen перед командой запуска долгоживущего процесса, а затем нажать комбинации клавиш Ctrl-A и Ctrl-D, чтобы отсоединиться от сеанса. Чтобы вновь подключиться к этому процессу, вам достаточно просто снова ввести команду `screen` и нажать клавишу Ctrl-A еще раз.

В примере 10.16 производится запуск программы tracd внутри сеанса screen. Как только процесс запустится, мы можем просто отсоединиться от сеанса, нажав комбинации клавиш Ctrl-A и Ctrl-D, если, конечно, предполагается, что позднее мы вновь будем подключаться к сеансу.

Пример 10.16. Запуск программ на языке Python под управлением программы screen

```
screen python2.4 /usr/bin/tracd --hostname=trac.example.com --port 8888
-r --single-env --auth=*,/home/noahgift/trac-instance/conf/
password,tracadminaccount /home/example/trac-instance/
```

Чтобы опять подключиться к этому сеансу, можно ввести команду:

```
[root@cent ~]# screen -r
There are several suitable screens on:
4797.pts-0.cent (Detached)
24145.pts-0.cent (Detached)
Type "screen [-d] -r [pid.]tty.host" to resume one of them.
```

Возможно, это не самый лучший подход для использования в рабочей среде, но для нужд разработки или личного использования он обладает определенными достоинствами.

Потоки выполнения в Python

Потоки выполнения нередко рассматриваются как необходимое зло. Несмотря на то, что многим потоки не нравятся, тем не менее, они позволяют решать задачи, когда приходится одновременно иметь дело сразу с несколькими вещами. Потоки выполнения – это не процессы, потому что они выполняются в пределах одного и того же процесса и совместно используют память процесса. Это одно из самых больших преимуществ и одновременно самый большой недостаток потоков. Преимущество заключается в том, что можно создавать в памяти структуры данных, которые будут доступны всем потокам выполнения без использования механизмов межпроцессных взаимодействий (IPC).

Но при работе с потоками имеются свои подводные камни. Часто тривиальная программа, состоящая из нескольких десятков строк программного кода, с введением потоков выполнения может стать чрезвычайно сложной. Многопоточные программы сложно отлаживать без использования всеобъемлющей трассировки, но и в этом случае отладка остается очень сложной процедурой, потому что вывод трассировочной информации может оказаться слишком объемным и запутанным. Один из авторов создал систему исследования центров обработки данных с помощью протокола SNMP, но реализовать в ней полную поддержку потоков выполнения оказалось очень непросто.

Однако существуют определенные стратегии, упрощающие создание многопоточных приложений, и реализация надежной библиотеки трассировки – одна из таких стратегий. При этом они могут оказаться очень удобным инструментом в решении сложных задач.

Знание основ программирования многопоточных приложений может оказаться полезным для системного администратора. Вот несколько примеров, когда потоки выполнения могут пригодиться в повседневной практике системного администратора: исследование локальной сети в автоматическом режиме, извлечение нескольких веб-страниц одновременно, нагрузочное тестирование сервера и выполнение сетевых операций.

Сохраняя верность принципу KISS, рассмотрим один из самых простых примеров использования нескольких потоков выполнения. Следует заметить, что для использования модуля threading необходимо понимание объектно-ориентированного программирования. Если у вас недостаточный опыт объектно-ориентированного программирования (ООП) или вообще его нет, тогда этот пример может оказаться непонятным для вас. В этом случае мы могли бы порекомендовать приобре-

сти книгу Марка Лутца (Mark Lutz) «Learning Python» (O'Reilly)[1] и познакомиться с некоторыми основами ООП, однако можно обратиться к главе 1 «Введение» в этой книге и попрактиковаться на некоторых примерах, которые там приводятся. В конечном счете, объектно-ориентированное программирование достойно того, чтобы изучать его.

Поскольку эта книга посвящена практическому применению языка Python, давайте перейдем непосредственно к примеру многопоточного приложения, где используются самые простые приемы многопоточного программирования. В этом простом многопоточном сценарии используется модуль threading. В сценарии устанавливается значение глобальной переменной, и затем переопределяется метод `run()` потока выполнения. Наконец, запускается пять потоков выполнения, каждый из которых выводит свой номер.

Во многих отношениях этот пример чрезмерно упрощен и имеет плохой дизайн, потому что в нем сразу несколько потоков используют одну и ту же глобальную переменную. Часто совместно с потоками лучше использовать очереди, так как они могут принять на себя всю сложность организации доступа к совместно используемым данным. Исходный текст сценария приводится в примере 10.17.

Пример 10.17. Простейший многопоточный сценарий

```
#не совсем правильно организован доступ к совместно используемым данным
import threading
import time
count = 1
class KissThread(threading.Thread):
    def run(self):
        global count
        print "Thread # %s: Pretending to do stuff" % count
        count += 1
        time.sleep(2)
        print "done with stuff"

for t in range(5):
    KissThread().start()
```

```
[ngift@Macintosh-6][H:10464][J:0]> python thread1.py
Thread # 1: Pretending to do stuff
Thread # 2: Pretending to do stuff
Thread # 3: Pretending to do stuff
Thread # 4: Pretending to do stuff
Thread # 5: Pretending to do stuff
done with stuff
done with stuff
done with stuff
```

[1] Марк Лутц. «Изучаем Python» – Пер. с англ. – СПб.: Символ-Плюс, 2009. – *Прим. перев.*

```
done with stuff
done with stuff
```

```
#common.py
import subprocess
import time

IP_LIST = [ 'google.com',
  'yahoo.com',
  'yelp.com',
  'amazon.com',
  'freebase.com',
  'clearink.com',
  'ironport.com' ]

cmd_stub = 'ping -c 5 %s'

def do_ping(addr):
    print time.asctime(), "DOING PING FOR", addr
    cmd = cmd_stub % (addr,)
    return subprocess.Popen(cmd, shell=True, stdout=subprocess.PIPE)
```

```
from common import IP_LIST, do_ping
import time

z = []
#for i in range(0, len(IP_LIST)):
for ip in IP_LIST:
    p = do_ping(ip)
    z.append((p, ip))

for p, ip in z:
    print time.asctime(), "WAITING FOR", ip
    p.wait()
    print time.asctime(), ip, "RETURNED", p.returncode
```

```
jmjones@dinkgutsy:thread_discuss$ python nothread.py
Sat Apr 19 06:45:43 2008 DOING PING FOR google.com
Sat Apr 19 06:45:43 2008 DOING PING FOR yahoo.com
Sat Apr 19 06:45:43 2008 DOING PING FOR yelp.com
Sat Apr 19 06:45:43 2008 DOING PING FOR amazon.com
Sat Apr 19 06:45:43 2008 DOING PING FOR freebase.com
Sat Apr 19 06:45:43 2008 DOING PING FOR clearink.com
Sat Apr 19 06:45:43 2008 DOING PING FOR ironport.com
Sat Apr 19 06:45:43 2008 WAITING FOR google.com
Sat Apr 19 06:45:47 2008 google.com RETURNED 0
Sat Apr 19 06:45:47 2008 WAITING FOR yahoo.com
Sat Apr 19 06:45:47 2008 yahoo.com RETURNED 0
Sat Apr 19 06:45:47 2008 WAITING FOR yelp.com
Sat Apr 19 06:45:47 2008 yelp.com RETURNED 0
Sat Apr 19 06:45:47 2008 WAITING FOR amazon.com
Sat Apr 19 06:45:57 2008 amazon.com RETURNED 1
Sat Apr 19 06:45:57 2008 WAITING FOR freebase.com
Sat Apr 19 06:45:57 2008 freebase.com RETURNED 0
```

```
Sat Apr 19 06:45:57 2008 WAITING FOR clearink.com
Sat Apr 19 06:45:57 2008 clearink.com RETURNED 0
Sat Apr 19 06:45:57 2008 WAITING FOR ironport.com
Sat Apr 19 06:46:58 2008 ironport.com RETURNED 0
```

В качестве оговорки к следующим примерам многопоточных сценариев следует заметить, что они являются достаточно сложными примерами и те же самые действия могут быть реализованы на основе применения функции `subprocess.Popen()`. Эта функция является лучшим выбором, когда требуется запустить группу процессов и дождаться их завершения. Если вам необходимо организовать взаимодействие с каждым процессом, то можно использовать функцию `subprocess.Popen()` в комплексе с потоками выполнения. Основная цель этих примеров – продемонстрировать, что многозадачность нередко требует уступок и компромиссов. Часто бывает очень трудно определить, какая модель лучше отвечает требованиям – потоки выполнения, процессы или асинхронные библиотеки, такие как stackless или twisted. Ниже приводится пример опроса с помощью утилиты ping большого массива IP-адресов.

Теперь, когда у нас имеется своеобразная программа «Hello World» для потоков выполнения, можно перейти к реализации сценария, который оценит любой системный администратор. Возьмем за основу наш сценарий и изменим его так, чтобы он опрашивал узлы в сети. Это можно считать начальным этапом на пути создания универсального инструмента для работы с сетью. Программный код сценария приводится в примере 10.18.

Пример 10.18. Многопоточная версия утилиты ping

```
#!/usr/bin/env python
from threading import Thread
import subprocess
from Queue import Queue

num_threads = 3
queue = Queue()
ips = ["10.0.1.1", "10.0.1.3", "10.0.1.11", "10.0.1.51"]
def pinger(i, q):
    """опрос подсети"""
    while True:
        ip = q.get()
        print "Thread %s: Pinging %s" % (i, ip)
        ret = subprocess.call("ping -c 1 %s" % ip,
                              shell=True,
                              stdout=open('/dev/null', 'w'),
                              stderr=subprocess.STDOUT)
        if ret == 0:
            print "%s: is alive" % ip
        else:
            print "%s: did not respond" % ip
```

```
        q.task_done()

for i in range(num_threads):

    worker = Thread(target=pinger, args=(i, queue))
    worker.setDaemon(True)
    worker.start()

for ip in ips:
    queue.put(ip)

print "Main Thread Waiting"
queue.join()
print "Done"
```

Когда мы запустили этот, достаточно простой фрагмент программного кода, мы получили следующий результат:

```
[ngift@Macintosh-6][H:10432][J:0]# python ping_thread_basic.py
Thread 0: Pinging 10.0.1.1
Thread 1: Pinging 10.0.1.3
Thread 2: Pinging 10.0.1.11
Main Thread Waiting
10.0.1.1: is alive
Thread 0: Pinging 10.0.1.51
10.0.1.3: is alive
10.0.1.51: is alive
10.0.1.11: did not respond
Done
```

Этот пример заслуживает того, чтобы разобрать его на понятные части, но сначала – небольшое пояснение. Пример разработки многопоточной версии утилиты ping с целью опроса подсети – это отличный способ продемонстрировать применение потоков. «Обычная» программа на языке Python, не использующая потоки выполнения, потребовала бы времени для своего выполнения N * (среднее время ожидания ответа на каждый запрос ping). Утилита ping может возвращать один из двух вариантов ответа: время отклика хоста и сообщение об истечении предельного времени ожидания. В типичной сети можно столкнуться с обоими вариантами.

Это означает, что на выполнение приложения, использующего утилиту ping для опроса хостов сети класса C, состоящей из 254 адресов, может потребоваться до 254 * (~ 3 секунды), что может составить до 12.7 минут. При использовании потоков это время можно уменьшить до нескольких секунд. Именно поэтому потоки имеют важное значение для разработки сетевых приложений. Теперь сделаем еще шаг и подумаем, какие условия могут встретиться в действительности. Сколько подсетей может существовать в типичном центре обработки данных? 20? 30? 50? Очевидно, что программа, выполняющая опрос последовательным способом, быстро теряет свою практическую ценность, и многопоточная версия становится идеальным выбором.

Теперь вернемся к нашему простому сценарию и рассмотрим некоторые особенности реализации. Первое, на что следует обратить внимание, – это импортируемые модули, в частности, наибольший интерес для нас представляют модули threading и Queue. Как уже отмечалось выше, разработка многопоточных приложений без использования очередей намного сложнее, и многим оказывается не под силу. Всякий раз, когда вам требуется прибегнуть к использованию потоков, желательно использовать модуль Queue. Почему? Этот модуль снижает потребность в явной реализации защиты данных с помощью мьютексов, потому что внутренние механизмы самих очередей обеспечивают необходимую защиту данных.

Представьте, что вы фермер/ученый, живущий в Средние века, и вы заметили, что вороны, которых часто называют «убийцами» (если интересно узнать, почему, обращайтесь в Википедию), атакуют ваши поля с зерновыми стаями по 20 или более особей.

Это очень умные птицы и их невозможно испугать, бросая камни, так как вы сможете бросать не чаще, чем один камень каждые 3 секунды, а численность стаи может достигать 50 особей. Чтобы отпугнуть всех ворон, может потребоваться до нескольких минут, но за этот промежуток времени урожаю может быть нанесен существенный ущерб. Как ученый, знающий математику, вы понимаете, что эта проблема легко разрешима. Вам нужно лишь набрать кучу камней и затем расставить работников, чтобы они могли одновременно брать камни из кучи и бросать в ворон.

Если следовать этой стратегии, 30 работников, выбирая камни из кучи, могли бы закидать камнями 50 ворон менее чем за 10 секунд. Это основа использования потоков и очередей в сценариях на языке Python. Вы нанимаете группу работников для выполнения какой-либо работы, и когда очередь опустеет, задание можно считать выполненным.

Очереди обеспечивают способ передачи заданий «группе» работников централизованным образом. Один из самых важных элементов нашей простой программы – это вызов метода `join()`. Описание метода `queue.join()` гласит следующее:

```
Namespace:   Interactive
File: /System/Library/Frameworks/Python.framework/Versions/2.5/lib/
python2.5/
  Queue.py
Definition:  Queue.Queue.join(self)
Docstring:
Blocks until all items in the Queue have been gotten and processed.
(Блокирует выполнение вызывающей программы, пока не будут обработаны
все элементы очереди)

The count of unfinished tasks goes up whenever an item is added to the
queue. The count goes down whenever a consumer thread calls task_done()
to indicate the item was retrieved and all work on it is complete.
```

```
(Всякий раз, когда в очередь добавляется новый элемент, увеличивается счетчик
невыполненных задач. Всякий раз, когда пользовательский поток вызывает метод
task_done(), чтобы показать, что изъятый из очереди элемент обработан,
счетчик уменьшается.)

When the count of unfinished tasks drops to zero, join() unblocks.
(Когда счетчик заданий уменьшается до нуля, метод join() возвращает
управление вызывающей программе.)
```

Метод `join()` представляет собой простой способ предотвратить завершение выполнения главного потока программы до того, как остальные потоки выполнения получат шанс завершить обработку элементов очереди. Возвращаясь к метафоре с фермером, главный поток можно сравнить с фермером, который собирает кучу камней и уходит, а работники выстраиваются в линию, готовясь бросать камни. Если в нашем примере закомментировать вызов метода `queue.join()`, отрицательные последствия этого не замедлят сказаться. Попробуем закомментировать вызов `queue.join()`:

```
print "Main Thread Waiting"
#Если закомментировать вызов метода join, главная программа завершится
#до того, как потоки получат возможность выполнить свою работу
#queue.join()
print "Done"
```

Теперь посмотрим, что выдаст наш замечательный сценарий. Взгляните на пример 10.19.

Пример 10.19. Пример, когда главный поток выполнения программы завершает работу раньше других потоков

```
[ngift@Macintosh-6][H:10189][J:0]# python ping_thread_basic.py
Main Thread Waiting
Done
Unhandled exception in thread started by
Error in sys.excepthook:

Original exception was:
```

Теперь, ознакомившись с теорией применения в сценариях потоков выполнения и очередей, пройдемся по программному коду шаг за шагом. В самом начале мы жестко определили несколько значений, которые в более универсальных программах обычно передаются в виде аргументов командной строки. Переменная `num_threads` содержит число рабочих потоков, переменная `queue` – это экземпляр очереди и, наконец, `ips` – это список IP-адресов, которые мы должны поместить в очередь:

```
num_threads = 3
queue = Queue()
ips = ["10.0.1.1", "10.0.1.3", "10.0.1.11", "10.0.1.51"]
```

Следующая функция выполняет основную работу в программе. Она вызывается каждым потоком и извлекает очередной IP-адрес из очереди.

Примечательно, что адреса выталкиваются из очереди в том же порядке, в каком они находятся в списке. Такая реализация позволяет извлекать элементы, пока очередь не опустеет. В конце цикла `while` **вызывается метод** `q.task_done()` **— это имеет важное значение, потому что он сообщает методу** `join()` **о том, что был обработан очередной элемент, извлеченный из очереди. Или, говоря простым языком, он сообщает о том, что задание выполнено. Посмотрим, что говорится в описании метода** `Queue.Queue.task_done()`**:**

```
File: /System/Library/Frameworks/Python.framework/Versions/2.5/lib/
python2.5/
  Queue.py
Definition:   Queue.Queue.task_done(self)
Docstring:
Indicate that a formerly enqueued task is complete.
(Свидетельствует о том, что очередное задание в очереди было выполнено.)

Used by Queue consumer threads. For each get() used to fetch a task,
a subsequent call to task_done() tells the queue that the processing
on the task is complete.
(Вызывается потоком-потребителем. Каждому вызову метода get(), используемому
для извлечения задания, должен соответствовать вызов метода task_done(),
который сообщает очереди, что задание выполнено.)

If a join() is currently blocking, it will resume when all items
have been processed (meaning that a task_done() call was received
for every item that had been put() into the queue).
(Метод join() вернет управление вызывающей программе, когда будут обработаны
все элементы (то есть для каждого элемента, помещенного в очередь вызовом
метода put(), будет вызван метод task_done()))

Raises a ValueError if called more times than there were items
placed in the queue.
(При вызове большее число раз, чем имелось элементов в очереди, возбуждает
исключение ValueError)
```

Из описания видно, что между методами `q.get()` **и** `q.task_done()` **существует взаимосвязь и в конечном счете они связаны с методом** `q.join()`**. Это практически начало, середина и конец истории:**

```
def pinger(i, q):
    """опрос подсети"""
    while True:
        ip = q.get()
        print "Thread %s: Pinging %s" % (i, ip)
        ret = subprocess.call("ping -c 1 %s" % ip,
                              shell=True,
                              stdout=open('/dev/null', 'w'),
                              stderr=subprocess.STDOUT)
        if ret == 0:
            print "%s: is alive" % ip
        else:
```

```
            print "%s: did not respond" % ip
        q.task_done()
```

Ниже мы используем простой цикл `for`, который управляет созданием группы потоков выполнения. Примечательно, что эта группа просто «сидит и ждет», пока что-нибудь не появится в очереди. В программе ничего не происходит, пока управление не достигнет следующего раздела.

В нашей программе кроется одна малозаметная хитрость, которая предохраняет программу от попадания в ловушку. Обратите внимание на вызов метода `setDaemon(True)`. Если этого не сделать перед вызовом метода `start()` потока, программа зависнет на неопределенный срок.

Причина практически незаметна на первый взгляд и заключается в том, что программа может завершить свою работу, только если потоки выполняются в режиме демонов. Вы могли заметить, что в функции `pinger()` используется бесконечный цикл. Поскольку поток, вызвавший такую функцию, никогда сам не завершится, нам пришлось объявить их потоками-демонами. Чтобы убедиться в справедливости вышесказанного, просто закомментируйте строку `worker.setDaemon(True)` и запустите программу. Заметим лишь, что без вызова этого метода программа будет крутиться вхолостую неопределенно продолжительное время. Обязательно проверьте это у себя, так как это поможет вам частично снять с процесса покров таинственности:

```
for i in range(num_threads):
    worker = Thread(target=pinger, args=(i, queue))
    worker.setDaemon(True)
    worker.start()
```

К этому моменту в нашей программе имеется группа готовых к работе потоков, ожидающих, пока мы дадим им задание. Как только мы поместим элементы в очередь, нашим потокам тут же будет послан сигнал, что можно извлечь задание из очереди, которое в данном случае заключается в том, чтобы опросить указанный IP-адрес:

```
for ip in ips:
    queue.put(ip)
```

Наконец, мы достигли критической строки, зажатой между двумя инструкциями `print`, которая в конечном счете управляет программой. Как уже говорилось ранее, вызвав метод очереди `join()`, главный поток программы становится в ожидание, пока не опустеет очередь заданий. Именно поэтому потоки и очередь напоминают шоколад с арахисовым маслом. Каждый из них обладает своей прелестью, но вместе они создают особый вкус.

```
print "Main Thread Waiting"
queue.join()
print "Done"
```

Чтобы лучше понять принципы использования потоков и очередей, нам нужно сделать еще один шаг вперед и добавить в наш пример еще одну группу потоков и еще одну очередь. В первом примере мы опрашивали с помощью утилиты ping список IP-адресов, которые извлекали из очереди. В следующем примере мы заставим первую группу потоков помещать IP-адреса, от которых был получен ответ, во вторую очередь.

После этого вторая группа потоков будет извлекать IP-адреса из второй очереди, производить опрос с помощью утилиты arping и возвращать IP-адреса вместе с MAC-адресами. Исходный текст примера приводится в примере 10.20.

Пример 10.20. Несколько очередей и несколько групп потоков

```
#!/usr/bin/env python
#Требуется Python2.5 или выше
from threading import Thread
import subprocess
from Queue import Queue
import re

num_ping_threads = 3
num_arp_threads = 3
in_queue = Queue()
out_queue = Queue()
ips = ["10.0.1.1", "10.0.1.3", "10.0.1.11", "10.0.1.51"]

def pinger(i, iq, oq):
    """опрос подсети"""
    while True:
        ip = iq.get()
        print "Thread %s: Pinging %s" % (i, ip)
        ret = subprocess.call("ping -c 1 %s" % ip,
                              shell=True,
                              stdout=open('/dev/null', 'w'),
                              stderr=subprocess.STDOUT)
        if ret == 0:
            #print "%s: is alive" % ip
            #поместить ответившие IP-адреса во вторую очередь
            oq.put(ip)
        else:
            print "%s: did not respond" % ip
        iq.task_done()

def arping(i, oq):
    """извлекает ответивший IP-адрес из очереди и получает MAC-адрес"""
    while True:
        ip = oq.get()
        p = subprocess.Popen("arping -c 1 %s" % ip,
                             shell=True,
                             stdout=subprocess.PIPE)
        out = p.stdout.read()
```

```
            #отыскать и извлечь MAC-адрес из потока стандартного вывода
            result = out.split()
            pattern = re.compile(":")
            macaddr = None
            for item in result:
                if re.search(pattern, item):
                    macaddr = item
            print "IP Address: %s | Mac Address: %s " % (ip, macaddr)
            oq.task_done()

    #Поместить IP-адреса в очередь
    for ip in ips:
        in_queue.put(ip)

    #Породить группу потоков, вызывающих утилиту ping
    for i in range(num_ping_threads):

        worker = Thread(target=pinger, args=(i, in_queue, out_queue))
        worker.setDaemon(True)
        worker.start()

    #Породить группу потоков, вызывающих утилиту arping
    for i in range(num_arp_threads):

        worker = Thread(target=arping, args=(i, out_queue))
        worker.setDaemon(True)
        worker.start()

    print "Main Thread Waiting"
    #Гарантировать, что программа не завершит работу,
    #пока не опустеют обе очереди
    in_queue.join()
    out_queue.join()

    print "Done"
```

После запуска этого сценария мы получили следующие результаты:

```
python2.5 ping_thread_basic_2.py
Main Thread Waiting
Thread 0: Pinging 10.0.1.1
Thread 1: Pinging 10.0.1.3
Thread 2: Pinging 10.0.1.11
Thread 0: Pinging 10.0.1.51
IP Address: 10.0.1.1 | Mac Address: [00:00:00:00:00:01]
IP Address: 10.0.1.51 | Mac Address: [00:00:00:80:E8:02]
IP Address: 10.0.1.3 | Mac Address: [00:00:00:07:E4:03]
10.0.1.11: did not respond
Done
```

Для реализации этого решения мы лишь немного расширили предыдущий пример, добавив в него еще одну группу потоков и еще одну очередь. Это достаточно важный прием, чтобы поместить его в свой арсенал, потому что модуль queue делает использование потоков более

простым и более безопасным делом. Можно даже сказать, что этот прием относится к разряду обязательных к применению.

Задержка выполнения потоков с помощью threading.Timer

В языке Python имеется еще одна особенность, имеющая отношение к потокам, которая может оказаться удобной при решении задач системного администрирования. Она существенно упрощает запуск функции с задержкой по времени, как показано в примере 10.21.

Пример 10.21. Таймер внутри потока

```
#!/usr/bin/env python
from threading import Timer
import sys
import time
import copy

#простейшая обработка ошибок
if len(sys.argv) != 2:
    print "Must enter an interval"
    sys.exit(1)

#функция, которая будет вызываться с задержкой по времени
def hello():
    print "Hello, I just got called after a %s sec delay" % call_time

#поток, который реализует задержку по времени
delay = sys.argv[1]
call_time = copy.copy(delay)  #копирование задержки для использования позднее
t = Timer(int(delay), hello)
t.start()

#показать, что программа не блокируется и продолжает работать
print "waiting %s seconds to run function" % delay
for x in range(int(delay)):
    print "Main program is still running for %s more sec" % delay
    delay = int(delay) - 1
    time.sleep(1)
```

Если запустить этот фрагмент, можно увидеть, что главный поток программы продолжает работу и при этом происходит отложенный вызов функции:

```
[ngift@Macintosh-6][H:10468][J:0]# python thread_timer.py 5
waiting 5 seconds to run function
Main program is still running for 5 more sec
Main program is still running for 4 more sec
Main program is still running for 3 more sec
Main program is still running for 2 more sec
Main program is still running for 1 more sec
Hello, I just got called after a 5 sec delay
```

Обработка событий в потоке

Эта книга предназначена для системных администраторов, поэтому давайте применим описанную выше методику для решения более практичной задачи. В этом примере мы возьмем на вооружение прием с отложенным запуском и объединим его с циклом ожидания событий, в котором будем проверять наличие расхождений в именах файлов между двумя каталогами. Мы могли бы пойти дальше и проверять время последнего изменения файлов, но, следуя принципу сохранения максимальной простоты примеров, мы посмотрим, как в этом цикле проверяется ожидаемое событие и как, в случае его появления, вызывается метод обработки.

Этот модуль легко можно преобразовать в более универсальный инструмент, но пока в примере 10.22 жестко определены каталоги, которые будут синхронизироваться с задержкой с помощью команды `rsync -av --delete`, если между ними будут обнаружены различия.

Пример 10.22. Инструмент синхронизации каталогов

```
#!/usr/bin/env python
from threading import Timer
import sys
import time
import copy
import os
from subprocess import call

class EventLoopDelaySpawn(object):
    """Класс обработки события, который запускает метод из потока задержки"""

    def __init__(self, poll=10,
                       wait=1,
                       verbose=True,
                       dir1="/tmp/dir1",
                       dir2="/tmp/dir2"):

        self.poll = int(poll)
        self.wait = int(wait)
        self.verbose = verbose
        self.dir1 = dir1
        self.dir2 = dir2

    def poller(self):
        """Интервал опроса"""
        time.sleep(self.poll)
        if self.verbose:
            print "Polling at %s sec interval" % self.poll

    def action(self):
        if self.verbose:
            print "waiting %s seconds to run Action" % self.wait
```

```
            ret = call("rsync -av --delete %s/ %s" % (self.dir1, self.dir2),
    shell=True)

        def eventHandler(self):
            #Если в каталогах имеются файлы с разными именами
            if os.listdir(self.dir1) != os.listdir(self.dir2):
                print os.listdir(self.dir1)
                t = Timer((self.wait), self.action)
                t.start()
                if self.verbose:
                    print "Event Registered"
            else:
                if self.verbose:
                    print "No Event Registered"

        def run(self):
            """Цикл проверки события с отложенным запуском обработчика"""
            try:
                while True:
                    self.eventHandler()
                    self.poller()

            except Exception, err:
                print "Error: %s " % err

            finally:
                sys.exit(0)

    E = EventLoopDelaySpawn()
    E.run()
```

Внимательные читатели могут заметить, что строго говоря, задержка здесь не является строго необходимой, и это действительно так. Но как бы то ни было, задержка может дать дополнительное преимущество. Если добавить задержку, скажем, на 5 секунд, можно было бы прервать выполнение потока в случае наступления другого события, например, в случае неожиданного удаления основного каталога. Задержка, реализованная в виде потока, представляет собой замечательный механизм создания операций с отложенным выполнением, которые можно отменить.

Процессы

Потоки – это не единственный способ использования многозадачности в языке Python. В действительности процессы обладают некоторыми преимуществами перед потоками, т. к. они, в отличие от потоков в языке Python, могут выполняться на разных процессорах. На самом деле, из-за наличия глобальной блокировки интерпретатора (Global Interpreter Lock, GIL) в каждый момент времени может выполняться только один поток управления и только на одном процессоре. Поэтому для решения «тяжеловесных» задач на языке Python потоки являются не

лучшим выбором. В таких случаях обработку лучше производить в разных процессах.

Процессы будут лучшим выбором, если для решения задачи потребуется задействовать несколько процессоров. Кроме того, существует множество библиотек, которые просто не могут работать с потоками управления. Например, текущая реализация библиотеки Net-SNMP для языка Python не является асинхронной, поэтому при необходимости выполнения параллельной обработки следует использовать ветвление процессов.

Потоки в приложении совместно используют одну и ту же область памяти, в то время как процессы полностью независимы друг от друга и для организации взаимодействия с процессом требуется приложить больше усилий. Обмен информацией с процессами с помощью каналов может оказаться непростым делом, но, к счастью, существует библиотека processing, которую мы здесь подробно рассмотрим. Идут разговоры о включении библиотеки processing в стандартную библиотеку языка Python, поэтому будет совсем нелишним познакомиться с ней поближе.

Ранее мы упоминали альтернативный метод создания множества процессов, основанный на использовании функции `subprocess.Popen()`. Во многих случаях это отличный выбор для организации параллельного выполнения программного кода. В главе 13 вы найдете пример, где этот прием используется для создания множества процессов dd.

Как упоминалось ранее, реализация параллельной обработки данных никогда не отличалась простотой. Этот пример можно было счесть неэффективным, потому что в нем используется функция `subprocess.Popen()`, вместо того чтобы с помощью модуля processing создавать дочерние процессы, в которых вызывать функцию `subprocess.call()`. Однако с точки зрения крупного приложения использование прикладного интерфейса, напоминающего очереди, имеет свои преимущества и сравнимо с примером многопоточного приложения, приведенного выше. В настоящее время идут разговоры об объединении модулей processing и Subprocess, потому что модулю Subprocess недостает возможности управления группой процессов, которая присутствует в модуле processing. Этот запрос был сделан в системе PEP (Python Enhancement Proposal – система приема предложений по улучшению Python) для модуля Subprocess: *http://www.python.org/dev/peps/pep-0324/*.

Модуль processing

Так что же это за модуль processing, о котором мы упомянули выше? На момент написания книги «processing – это пакет для языка Python, который поддерживает возможность порождения процессов с помощью API модуля threading из стандартной библиотеки...». Одна из за-

мечательных особенностей модуля processing заключается в том, что он до определенной степени соответствует прикладному интерфейсу модуля threading. Это означает, что вам не придется изучать новый API, чтобы порождать новые процессы вместо потоков. Подробнее о модуле processing можно прочитать по адресу: *http://pypi.python.org/pypi/processing*.

Теперь, когда мы получили некоторые сведения о модуле processing, рассмотрим пример 10.23.

Пример 10.23. Введение в модуль processing

```
#!/usr/bin/env python
from processing import Process, Queue
import time

def f(q):
    x = q.get()
    print "Process number %s, sleeps for %s seconds" % (x,x)
    time.sleep(x)
    print "Process number %s finished" % x

q = Queue()

for i in range(10):
    q.put(i)
    i = Process(target=f, args=[q])
    i.start()

print "main process joins on queue"
i.join()
print "Main Program finished"
```

Запустив этот фрагмент, мы получили следующее:

```
[ngift@Macintosh-7][H:11199][J:0]# python processing1.py
Process number 0, sleeps for 0 seconds
Process number 0 finished
Process number 1, sleeps for 1 seconds
Process number 2, sleeps for 2 seconds
Process number 3, sleeps for 3 seconds
Process number 4, sleeps for 4 seconds
main process joins on queue
Process number 5, sleeps for 5 seconds
Process number 6, sleeps for 6 seconds
Process number 8, sleeps for 8 seconds
Process number 7, sleeps for 7 seconds
Process number 9, sleeps for 9 seconds
Process number 1 finished
Process number 2 finished
Process number 3 finished
Process number 4 finished
Process number 5 finished
Process number 6 finished
```

```
Process number 7 finished
Process number 8 finished
Process number 9 finished
Main Program finished
```

Все, что делает эта программа, это предписывает каждому процессу приостановиться на количество секунд, соответствующее порядковому номеру процесса. Как видите, интерфейс модуля прост и понятен.

Теперь, когда у нас имеется своего рода программа «Hello World», демонстрирующая использование модуля processing, можно создать что-нибудь более интересное. Если вы помните, в разделе с описанием потоков управления мы создали простой многопоточный сценарий, выполняющий опрос подсети. Поскольку прикладной интерфейс модуля processing очень напоминает интерфейс модуля threading, мы можем реализовать практически идентичный сценарий, используя процессы вместо потоков управления, как показано в примере 10.24.

Пример 10.24. Утилита ping на основе процессов

```
#!/usr/bin/env python
from processing import Process, Queue, Pool
import time
import subprocess
from IPy import IP
import sys

q = Queue()
ips = IP("10.0.1.0/24")
def f(i,q):
    while True:
        if q.empty():
            sys.exit()
        print "Process Number: %s" % i
        ip = q.get()
        ret = subprocess.call("ping -c 1 %s" % ip,
                        shell=True,
                        stdout=open('/dev/null', 'w'),
                        stderr=subprocess.STDOUT)
        if ret == 0:
            print "%s: is alive" % ip
        else:
           print "Process Number: %s didn't find a response for %s " % (i, ip)

for ip in ips:
    q.put(ip)

#q.put("192.168.1.1")

for i in range(50):
    p = Process(target=f, args=[i,q])
    p.start()

print "main process joins on queue"
```

```
p.join()
print "Main Program finished"
```

Этот сценарий удивительно похож на многопоточную версию, которая рассматривалась ранее. Если запустить этот сценарий, мы увидим примерно следующее:

```
[обрезано]
10.0.1.255: is alive
Process Number: 48 didn't find a response for 10.0.1.216
Process Number: 47 didn't find a response for 10.0.1.217
Process Number: 49 didn't find a response for 10.0.1.218
Process Number: 46 didn't find a response for 10.0.1.219
Main Program finished
[обрезано]
[ngift@Macintosh-7][H:11205][J:0]#
```

Этот пример требует дополнительных пояснений. Хотя прикладные интерфейсы модулей очень похожи, между ними все-таки есть некоторые отличия. Обратите внимание, что каждый из процессов запускается внутри бесконечного цикла, где выполняется извлечение элементов из очереди. Чтобы сообщить процессу о том, что он должен завершить работу, мы добавили условную инструкцию, которая проверяет, не опустела ли очередь. Каждый из 50 дочерних процессов сначала проверяет, не опустела ли очередь, и если в очереди нет элементов, процесс сам «убивает» себя, вызывая функцию `sys.exit()`.

Если в очереди еще имеются элементы, то процесс благополучно извлекает очередной элемент, в данном случае – IP-адрес, и приступает к выполнению своего задания, то есть выполняет опрос заданного IP-адреса с помощью утилиты ping. Главная программа использует метод `join()`, точно так же, как и версия сценария, реализованная на основе потоков, и ожидает, пока очередь не опустеет. После того как все рабочие процессы завершатся, и очередь опустеет, следующая ниже инструкция `print` сообщит о завершении программы.

Благодаря похожести прикладного интерфейса модуль processing использовать так же просто, как и модуль threading. В главе 7 мы обсуждали практическую реализацию на основе модуля processing сценария, использующего библиотеку Net-SNMP, которая по своей природе не является асинхронным расширением для языка Python.

Планирование запуска процессов Python

Теперь, когда мы рассмотрели разнообразные способы работы с процессами в языке Python, нам следует поговорить о способах планирования выполнения этих процессов. Для запуска программ, написанных на языке Python, вполне подходит старый добрый планировщик cron.

Одна из новых замечательных особенностей планировщика cron, имеющегося в большинстве POSIX-совместимых систем, заключается во введении каталогов планирования. Это и есть то, из-за чего мы используем cron, так как достаточно просто скопировать сценарий на языке Python в один из четырех каталогов по умолчанию: */etc/cron.daily*, */etc/cron.hourly*, */etc/cron.monthly* и */etc/cron.weekly*.

Достаточно много системных администраторов хотя бы раз в своей жизни обеспечивали возможность отправки отчета об использовании дискового пространства по электронной почте. Для этого вы просто помещаете в каталог */etc/cron.daily* сценарий на языке Bash, который содержит примерно следующее:

```
df -h | mail -s "Nightly Disk Usage Report" staff@example.com
```

Сохранив сценарий под именем */etc/cron.daily/diskusage.sh*, вы начинаете каждый день получать по электронной почте отчеты, имеющие примерно такой вид:

```
From:    guru-python-sysadmin@example.com
Subject: Nightly Disk Usage Report
Date:    February 24, 2029 10:18:57 PM EST
To:      staff@example.com

Filesystem            Size  Used Avail Use% Mounted on
/dev/hda3              72G   16G   52G  24% /
/dev/hda1              99M   20M   75M  21% /boot
tmpfs                1010M     0 1010M   0% /dev/shm
```

Но существует лучший путь. Даже для реализации заданий планировщика cron можно использовать преимущества языка Python вместо Bash или Perl. В действительности планировщик cron и Python прекрасно работают вместе. Давайте возьмем сценарий на языке Bash и реализуем его на языке Python, как показано в примере 10.25.

Пример 10.25. Отсылка ежедневного отчета об использовании дискового пространства по электронной почте

```
import smtplib
import subprocess
import string

p = subprocess.Popen("df -h", shell=True, stdout=subprocess.PIPE)
MSG = p.stdout.read()
FROM = "guru-python-sysadmin@example.com"
TO = "staff@example.com"
SUBJECT = "Nightly Disk Usage Report"
msg = string.join((
    "From: %s" % FROM,
    "To: %s" % TO,
    "Subject: %s" % SUBJECT,
    "",
    MSG), "\r\n")
```

```
server = smtplib.SMTP('localhost')
server.sendmail(FROM, TO, msg)
server.quit()
```

Это тривиальный рецепт создания автоматизированного отчета об использовании дискового пространства на базе cron, но он прекрасно подойдет для решения множества задач. Теперь подробнее рассмотрим, что делает этот небольшой фрагмент программного кода на языке Python. В первую очередь, с помощью `subprocess.Popen()` выполняется чтение потока стандартного вывода команды df. Затем создаются переменные для заполнения полей `From`, `To` и `Subject`. Затем объединением всех строк создается сообщение. Это самая сложная часть сценария. В заключение мы указываем имя `localhost` в качестве имени сервера исходящей почты и передаем переменные, созданные ранее, функции `server.sendmail()`.

Для того чтобы использовать такой сценарий, его обычно помещают в файл */etc/cron.daily/nightly_disk_report.py*.

Если вы еще только начинаете знакомиться с языком Python, можете использовать этот программный код как шаблон для быстрого создания работающих сценариев. В главе 4 мы немного подробнее обсуждали вопрос создания сообщений электронной почты, поэтому за дополнительной информацией вы можете обращаться к этой главе.

Запуск демона

Работа с демонами – это данность для любого, кто потратил на операционную систему UNIX больше времени, чем необходимо для беглого знакомства. Демоны выполняют практически любые операции – от обработки запросов до пересылки файлов на принтер (например, `lpd`), приема запросов HTTP и передачи файлов (например, демон `httpd` вебсервера Apache).

Так что же такое демон? Часто под демоном понимают выполняющийся в фоновом режиме процесс, который не имеет управляющего терминала. Если вы знакомы с механизмом управления заданиями в UNIX, у вас может сложиться мнение, что добавление символа `&` в конце команды создаст демона. Или нажатие комбинации Ctrl-Z после запуска процесса с последующей командой `bg` создаст демона. В обоих случаях вы получите фоновые процессы, но ни один из этих способов не отрывает процесс от командной оболочки и не лишает его управляющего терминала (возможно, принадлежащего процессу командной оболочки). Итак, три основных признака демона: выполнение в фоновом режиме, отсутствие связи с процессом, запустившим его, и отсутствие управляющего терминала. Процесс, запущенный в фоновом режиме при помощи механизма управления заданиями, отвечает только первому требованию.

Ниже приводится фрагмент программного кода, в котором определяется функция daemonize(). Она превращает вызывающий ее процесс в демона – в том смысле, в каком говорилось в предыдущем параграфе. Эта функция была взята из рецепта «Forking a Daemon Process on Unix», который приводится во втором издании книги Дэвида Ашера (David Asher) «Python Cookbook» (O'Reilly) на страницах 388-389. Этот программный код достаточно близко следует рекомендациям, которые предлагает Ричард Стивенс (Richard Stevens) в своей книге «UNIX Network Programming: The Sockets Networking API» (O'Reilly) в качестве «правильного» способа создания демона. Для тех, кто не знаком с книгой Стивенса, заметим, что она обычно рассматривается как справочник по сетевому программированию, а также как руководство по созданию демонов в UNIX. Исходный текст функции приводится в примере 10.26.

Пример 10.26. Функция daemonize

```
import sys, os
def daemonize (stdin='/dev/null', stdout='/dev/null', stderr='/dev/null'):
    # Выполнить первое ветвление процесса.
    try:
        pid = os.fork()
        if pid > 0:
            sys.exit(0) # Первый родительский процесс завершает работу.
    except OSError, e:
        sys.stderr.write("fork #1 failed: (%d) %s\n" % (e.errno, e.strerror))
        sys.exit(1)
    # Отключиться от родительского окружения.
    os.chdir("/")
    os.umask(0)
    os.setsid()
    # Выполнить второе ветвление.
    try:
        pid = os.fork()
        if pid > 0:
            sys.exit(0) # Второй родительский процесс завершает работу.
    except OSError, e:
        sys.stderr.write("fork #2 failed: (%d) %s\n" % (e.errno, e.strerror))
        sys.exit(1)
    # Теперь процесс стал демоном,
    # выполнить перенаправление стандартных дескрипторов.
    for f in sys.stdout, sys.stderr: f.flush()
    si = file(stdin, 'r')
    so = file(stdout, 'a+')
    se = file(stderr, 'a+', 0)
    os.dup2(si.fileno(), sys.stdin.fileno())
    os.dup2(so.fileno(), sys.stdout.fileno())
    os.dup2(se.fileno(), sys.stderr.fileno())
```

Первое, что делает эта функция, – с помощью функции fork() производит ветвление процесса. В этом случае создается копия работающего

процесса, и эта копия рассматривается как «дочерний» процесс, а оригинал – как «родительский» процесс. После создания копии родительский процесс может завершить свою работу. Для этого проверяется идентификатор процесса `pid` после ветвления. Если идентификатор представлен положительным числом, это означает, что выполняется родительский процесс. Если вы никогда не программировали ветвление процессов с помощью функции `fork()`, это может показаться вам странным. После возврата из функции `os.fork()` в системе появляется две копии одного и того же работающего процесса. Обе они проверяют код, возвращаемый функцией `fork()`, который в дочернем процессе будет иметь значение **0**, а в родительском процессе – соответствовать идентификатору процесса. Любое ненулевое значение возвращается только родительскому процессу, который должен завершить работу. Если здесь возникло исключение, процесс просто завершается. Если этот сценарий вызывается из интерактивной командной оболочки (такой как Bash), вы в этот момент вернетесь в строку приглашения к вводу, потому что тот процесс, который вы запускали, только что завершил работу. Но дочерний процесс продолжает свою работу.

Затем процесс изменяет рабочий каталог на / (`os.chdir("/")`), устанавливает маску в значение `0` (`os.umask(0)`) и создает новый сеанс (`os.setsid()`). Изменение каталога на / переводит процесс демона в каталог, который всегда существует. Дополнительное преимущество, которое дает операция перехода в каталог /, заключается в том, что долгоживущий процесс не будет препятствовать возможности отмонтировать файловую систему, если получилось, что он был запущен из каталога в файловой системе, которую вы пожелаете отмонтировать. Затем процесс изменяет свою маску режима создания файлов на маску с более широкими правами. Если демон должен создавать файлы с правами доступа для группы, унаследованная маска с более ограниченными правами может давать отрицательный эффект. Последнее из этих трех действий (`os.setsid()`), пожалуй, наименее знакомо большинству читателей. Функция `setsid()` выполняет множество действий. Во-первых, она делает процесс лидером нового сеанса. Далее, она делает процесс лидером новой группы процессов. Наконец, пожалуй, самое важное для демонов – она лишает процесс управляющего терминала. Факт отсутствия управляющего терминала означает, что процесс не может пасть жертвой неумышленных (или преднамеренных) операций с механизмом управления заданиями с какого-либо терминала. Для долгоживущих процессов, таких как демоны, очень важно исключить возможность прерывания работы.

Но самое интересное на этом не заканчивается. После вызова функции `os.setsid()` производится повторное ветвление. Первое ветвление процесса и вызов функции `setsid()` лишь готовят почву для второго ветвления – они отсоединяют процесс от какого-либо управляющего терминала и делают его лидером сеанса. Второе ветвление означает, что получившийся процесс не может быть лидером сеанса, а также то, что

процесс не может приобрести управляющий терминал. Второе ветвление не является обязательной процедурой и выполняется больше из предосторожности. Без второго ветвления процесс мог бы приобрести управляющий терминал, открыв любое терминальное устройство без флага `O_NOCTTY`.

Последнее, что делает функция, – выполняет очистку файлов и производит их реорганизацию. Выталкивается информация в стандартных потоках вывода и вывода сообщений об ошибках (`sys.stdout` и `sys.stderr`). Тем самым гарантируется вывод информации, которая еще не была выведена. Функция `daemonize()` позволяет вызывающей программе определять файлы, которые будут играть роль потоков `stdin`, `stdout` и `stderr`. По умолчанию в качестве всех трех файлов используется устройство `/dev/null`. В этом месте функция принимает либо указанные пользователем файлы, либо значения по умолчанию и устанавливает стандартный ввод, стандартный вывод и стандартный вывод сообщений об ошибках в соответствие этим файлам.

Как можно использовать функцию `daemonize()`? Предположим, что у нас имеется программный код демона в виде сценария `daemonize.py`. В примере 10.27 приводится пример сценария, использующего эту функцию.

Пример 10.27. Использование функции daemonize( )

```
from daemonize import daemonize
import time
import sys

def mod_5_watcher():
    start_time = time.time()
    end_time = start_time + 20
    while time.time() < end_time:
        now = time.time()
        if int(now) % 5 == 0:
            sys.stderr.write('Mod 5 at %s\n' % now)
        else:
            sys.stdout.write('No mod 5 at %s\n' % now)
        time.sleep(1)

if __name__ == '__main__':
    daemonize(stdout='/tmp/stdout.log', stderr='/tmp/stderr.log')
    mod_5_watcher()
```

Этот сценарий сначала переходит в режим демона, определяя при этом, что в качестве стандартного вывода будет использоваться файл */tmp/stdout.log*, а в качестве стандартного вывода сообщений об ошибках будет использоваться файл */tmp/stderr.log*. Затем в течение 20 секунд, с интервалами в 1 секунду между проверками, он отслеживает текущее время. Если время, выраженное в секундах, делится на пять без остатка, производится запись сообщения в поток стандартного вывода сообщений об ошибках. Если время не делится на пять, произво-

дится запись сообщения в поток стандартного вывода. Так как процесс использует файлы */tmp/stdout.log* и */tmp/stderr.log* в качестве стандартного вывода и стандартного вывода сообщений об ошибках, соответственно, то мы имеем возможность наблюдать эти сообщения после запуска этого примера...

Сразу же после запуска сценария происходит возврат в строку приглашения к вводу:

```
jmjones@dinkgutsy:code$ python use_daemonize.py
jmjones@dinkgutsy:code$
```

И ниже приводится результат работы примера:

```
jmjones@dinkgutsy:code$ cat /tmp/stdout.log
No mod 5 at 1207272453.18
No mod 5 at 1207272454.18
No mod 5 at 1207272456.18
No mod 5 at 1207272457.19
No mod 5 at 1207272458.19
No mod 5 at 1207272459.19
No mod 5 at 1207272461.2
No mod 5 at 1207272462.2
No mod 5 at 1207272463.2
No mod 5 at 1207272464.2
No mod 5 at 1207272466.2
No mod 5 at 1207272467.2
No mod 5 at 1207272468.2
No mod 5 at 1207272469.2
No mod 5 at 1207272471.2
No mod 5 at 1207272472.2
jmjones@dinkgutsy:code$ cat /tmp/stderr.log
Mod 5 at 1207272455.18
Mod 5 at 1207272460.2
Mod 5 at 1207272465.2
Mod 5 at 1207272470.2
```

Это действительно очень простой пример написания демона, но мы надеемся, что он наглядно демонстрирует некоторые базовые понятия. Вы можете использовать функцию `daemonize()` для создания демона, который следит за состоянием каталога, выполняет мониторинг сети, сетевых серверов и всего, что угодно, и работает продолжительное (или неопределенно продолжительное) время.

В заключение

Хотелось бы надеяться, что эта глава продемонстрировала, насколько широкими и мощными возможностями обладает язык Python для работы с процессами. В языке Python реализован весьма изящный и сложный прикладной интерфейс для работы с потоками выполнения, но при этом всегда полезно помнить о существовании GIL. Если

вы связаны с вводом-выводом, тогда зачастую эта блокировка не является проблемой, но если вам требуется загрузить работой несколько процессоров, то лучше будет использовать несколько процессов. Некоторые считают, что процессы предпочтительнее, чем потоки, даже если бы не было блокировки GIL. Главная причина появления такого мнения состоит в том, что отладка многопоточного программного кода может превратиться в кошмар.

Наконец, будет совсем не лишним поближе познакомиться с модулем subprocess, если вы с ним еще не знакомы. Subprocess – это универсальный модуль, построенный по принципу «все в одном», предназначенный для работы с... ну. пусть будет, с подпроцессами.

11

Создание графического интерфейса

Когда информированные люди перечисляют обязанности системного администратора, разработка программ с графическим интерфейсом пользователя (ГИП) обычно не входит в их число. Тем не менее, бывают моменты, когда администратору просто *необходимо* создать приложение с графическим интерфейсом или создание такого приложения сможет существенно облегчить ему жизнь. Здесь мы рассматриваем идею графического интерфейса в широком смысле, подразумевая как традиционные приложения – с графическим интерфейсом на базе таких библиотек, как GTK или Qt, так и приложения с веб-интерфейсом.

В этой главе все наше внимание будет сосредоточено на использовании библиотек PyGTK, curses и веб-платформы Django. Для начала мы рассмотрим основы создания графического интерфейса, затем перейдем к исследованию очень простого приложения, использующего библиотеку PyGTK, а потом напишем то же самое приложение с использованием curses и Django. Наконец, разберем, как с помощью Django и небольшого объема программного кода можно написать приложение для работы с базой данных, имеющее привлекательный интерфейс.

Теория создания графического интерфейса

Когда создается консольная утилита, предполагается, что она будет выполнять все необходимые действия без вмешательства пользователя. Такое положение дел имеет место, когда сценарии запускаются с помощью таких планировщиков заданий, как `cron` и `at`. Но когда создается утилита с графическим интерфейсом, предполагается, что пользователь должен будет что-то вводить, чтобы эта утилита могла выполнить свою работу. Вспомните свой опыт работы с графическими приложениями, такими как веб-броузеры, клиенты электронной почты и текстовые процессоры. Вы *запускаете* приложение некоторым способом.

Приложение выполняет некоторые действия по инициализации, возможно, загружает какие-нибудь конфигурационные файлы и переводит себя в некоторое определенное состояние. Но после этого приложение просто ждет, пока пользователь сделает что-нибудь. Конечно, существуют примеры приложений, выполняющих некоторые действия самостоятельно, как, например, Firefox автоматически проверяет наличие обновлений без явного требования или согласия пользователя, но это уже другая история.

Чего ожидает приложение? Как оно узнает, что делать, когда пользователь предпримет какое-либо действие? Приложение ожидает, пока произойдет *событие*. Событие – это то, что происходит в пределах приложения с одним из визуальных элементов управления, например нажатие кнопки или выбор флажка. И приложение «знает», что делать, когда происходят такие события, потому что программист связывает определенные события с определенными фрагментами программного кода. «Фрагменты программного кода», связанные с определенными событиями, называют *обработчиками событий*. Одно из предназначений библиотек, на базе которых создается графический интерфейс, заключается в том, чтобы вызвать правильный обработчик события, когда происходит некоторое событие. Если быть более точным, библиотека графического интерфейса реализует «цикл событий», в пределах которого выполняется проверка поступления новых событий, и, когда события происходят, обрабатывает их соответствующим способом.

Поведение приложения управляется событиями. Когда вы пишете приложение с графическим интерфейсом, вы сами решаете, как приложение должно реагировать на те или иные действия пользователя. Вы создаете обработчики событий, которые будут вызываться библиотекой графического интерфейса при возбуждении событий пользователем.

Это описание соответствует приложениям, а как сформировать сам интерфейс? То есть как создавать кнопки, текстовые поля ввода, метки и флажки в приложении? Ответ на этот вопрос зависит от используемого инструментария. Графический интерфейс можно создать с помощью специальной программы-построителя графического интерфейса, входящей в состав выбранной вами библиотеки. Построитель графического интерфейса позволяет разместить в форме будущего приложения различные визуальные компоненты, такие как кнопки, метки, флажки и другие. Например, если вы работаете в операционной системе Mac OS X и выбрали в качестве основы библиотеку Cocoa, то для размещения графических компонентов можно воспользоваться программой Interface Builder. Или, если вы выбрали PyGTK в Linux, можно воспользоваться программой Glade. Или, если вы выбрали PyQt, можно прибегнуть к помощи программы Qt Designer.

Построители графического интерфейса удобны в использовании, но иногда у вас может появиться желание иметь более полный контроль

над графическим интерфейсом, чем может предложить программа-построитель. В таких случаях будет совсем несложно создать графический интерфейс «вручную», написав немного программного кода. В библиотеке PyGTK каждому типу графических элементов соответствует свой класс на языке Python. Например, окно – это объект класса `gtk.Window`. Кнопка – это объект класса `gtk.Button`. Чтобы создать простое приложение с графическим интерфесом, которое имеет окно и кнопку, вы создаете экземпляры классов `gtk.Window` и `gtk.Button` и добавляете кнопку в окно. Если необходимо, чтобы по щелчку на кнопке выполнялись некоторые действия, вы должны определить обработчик события «щелчка» на кнопке.

Создание простого приложения PyGTK

Мы создадим простой сценарий, использующий уже упоминавшиеся классы `gtk.Window` и `gtk.Button`. Ниже приводится исходный текст простого приложения с графическим интерфейсом, которое не делает ничего полезного, но демонстрирует некоторые основные принципы создания программ с графическим интерфейсом.

Прежде чем можно будет опробовать этот пример или написать свое собственное приложение на базе библиотеки PyGTK, вам необходимо установить ее. В современных дистрибутивах Linux установка выполняется достаточно просто. Она выполняется просто даже для Windows. Если вы пользуетесь дистрибутивом Ubuntu, эта библиотека должна быть уже установлена. Если для вашей платформы нет готового двоичного дистрибутива, то установка может оказаться достаточно сложной. Исходный текст приложения приводится в примере 11.1.

Пример 11.1. Простое приложение PyGTK с одним окном и с одной кнопкой

```
#!/usr/bin/env python

import pygtk
import gtk
import time

class SimpleButtonApp(object):
    """Это простое приложение PyGTK с одним окном и с одной кнопкой.
    После щелчка на кнопке на ней отображается текущее время.
    """

    def __init__(self):
        #Главное окно приложения
        self.window = gtk.Window(gtk.WINDOW_TOPLEVEL)

        #Так "регистрируется" обработчик события. Этот вызов
        #предписывает главному циклу gtk вызвать self.quit(),
        #когда окно "посылает" сигнал "destroy".
        self.window.connect("destroy", self.quit)
```

```
        #Надпись на кнопке "Click Me"
        self.button = gtk.Button("Click Me")

        #Регистрация другого обработчика события. На этот раз, когда
        #кнопка "посылает" сигнал "clicked", будет вызываться метод
        #'update_button_label'.
        self.button.connect("clicked", self.update_button_label, None)

        #Окно - это контейнер. Метод "add" вставляет кнопку в окно.
        self.window.add(self.button)

        #Этот вызов делает кнопку видимой, но она не станет видимой,
        #пока не станет видимым содержащий ее контейнер.
        self.button.show()

        #Сделать контейнер видимым
        self.window.show()

    def update_button_label(self, widget, data=None):
        """Помещает на кнопку надпись с текущим временем
        Это обработчик события 'clicked' кнопки
        """
        self.button.set_label(time.asctime())

    def quit(self, widget, data=None):
        """Останавливает главный цикл событий gtk
        Когда пользователь закрывает окно, оно исчезнет, но,
        если не остановить главный цикл событий gtk, приложение
        продолжит работу, хотя все будет выглядеть так, как будто
        ничего не происходит
        """
        gtk.main_quit()

def main(self):
    """Запуск главного цикла событий gtk"""
    gtk.main()

if __name__ == "__main__":
    s = SimpleButtonApp()
    s.main()
```

Самое первое, на что вы наверняка обратили внимание в этом примере, это то, что главный класс приложения наследует класс `object`, а не какой-нибудь класс GTK. Приложение с графическим интерфейсом на базе PyGTK не обязательно должно быть реализовано в объектно-ориентированном стиле. Безусловно, вам придется создавать экземпляры классов, но вы не обязаны создавать собственные классы. Однако для чего-то большего, чем тривиальный пример, такой как этот, мы настоятельно рекомендуем создавать собственные классы. Главное преимущество такого подхода к созданию приложений с графическим интерфейсом заключается в том, что все визуальные компоненты (окна, кнопки, флажки) будут прикреплены к одному и тому же объекту, что обеспечит прямой доступ к ним из любой части приложения.

Т. к. мы предпочли создать свой собственный класс, то сразу же начнем с того, что происходит в конструкторе (метод `__init__()`). Фактически, почти все, что делает это приложение, сосредоточено в конструкторе. Этот пример содержит подробные комментарии, поэтому мы не будем дублировать все пояснения здесь, а отметим наиболее важные моменты. В конструкторе создаются два объекта графического интерфейса: `gtk.Window` и `gtk.Button`. Затем кнопка помещается в окно, так как окно – это контейнерный объект. Мы также создали обработчики событий `destroy` и `clicked`, порождаемых окном и кнопкой соответственно. После запуска приложения на экране появляется окно с кнопкой, имеющей надпись «Click Me» (щелкни здесь). Каждый раз, когда производится щелчок на кнопке, надпись на кнопке обновляется и отображает текущее время. На рис. 11.1 и 11.2 приводится внешний вид приложения до и после щелчка на кнопке.

Рис. 11.1. *Простое приложение PyGTK – до щелчка на кнопке*

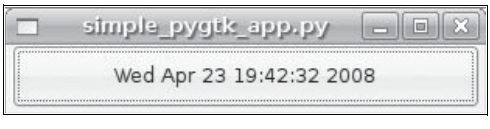


Рис. 11.2. *Простое приложение PyGTK – после щелчка на кнопке*

Создание приложения PyGTK для просмотра файла журнала веб-сервера Apache

Теперь, когда мы рассмотрели основы создания графического интерфейса в общем и с использованием PyGTK, перейдем к примеру, который с помощью PyGTK реализует нечто более полезное – рассмотрим создание приложения для просмотра содержимого файла журнала веб-сервера Apache. Это приложение будет обладать следующими функциональными возможностями:

- Позволит выбирать и открывать требуемый файл журнала
- Будет отображать номер строки, имя удаленного хоста, код состояния и количество переданных байтов
- Позволит сортировать строки по их номерам, по именам удаленных хостов, коду состояния или количеству переданных байтов

Этот пример основан на программном коде, выполняющем анализ файла журнала Apache, который мы написали в главе 3. Исходный текст приложения приводится в примере 11.2.

Пример 11.2. Приложение PyGTK для просмотра файла журнала веб-сервера Apache

```
#!/usr/bin/env python

import gtk
from apache_log_parser_regex import dictify_logline

class ApacheLogViewer(object):
    """Программа просмотра файла журнала веб-сервера Apache, которая
    позволяет сортировать информацию по разным полям данных"""

    def __init__(self):
        #Главное окно приложения
        self.window = gtk.Window(gtk.WINDOW_TOPLEVEL)
        self.window.set_size_request(640, 480)
        self.window.maximize()

        #Остановить цикл событий при закрытии окна
        self.window.connect("destroy", self.quit)

        #VBox – это контейнер, позволяющий добавлять в него визуальные
        #компоненты. Используется в первую очередь для обеспечения
        #определенного порядка расположения компонентов
        self.outer_vbox = gtk.VBox()

        #Панель инструментов с кнопками открытия журнала
        #и завершения приложения
        self.toolbar = gtk.Toolbar()

        #Создать кнопки открытия файла и завершения приложения
        #и пиктограммы добавить кнопки на панель инструментов
        #связать кнопки с соответствующими обработчиками событий
        open_icon = gtk.Image()
        quit_icon = gtk.Image()
       open_icon.set_from_stock(gtk.STOCK_OPEN, gtk.ICON_SIZE_LARGE_TOOLBAR)
       quit_icon.set_from_stock(gtk.STOCK_QUIT, gtk.ICON_SIZE_LARGE_TOOLBAR)
        self.open_button = gtk.ToolButton(icon_widget=open_icon)
        self.quit_button = gtk.ToolButton(icon_widget=quit_icon)
        self.open_button.connect("clicked", self.show_file_chooser)
        self.quit_button.connect("clicked", self.quit)
        self.toolbar.insert(self.open_button, 0)
        self.toolbar.insert(self.quit_button, 1)

        #Элемент управления для выбора открываемого файла
        self.file_chooser = gtk.FileChooserWidget()
        self.file_chooser.connect("file_activated", self.load_logfile)

        #ListStore используется для представления данных, имеющих вид
        #списка. Элемент ListStore будет хранить табличные данные в виде:
        #номер_строки, имя_хоста, статус, переданные_байты, текст_записи
        self.loglines_store = gtk.ListStore(int, str, str, int, str)
```

```
        #связать дерево с данными...
        self.loglines_tree = gtk.TreeView(model=self.loglines_store)

        #...и добавить надписи в заголовки колонок
        self.add_column(self.loglines_tree, 'Line Number', 0)
        self.add_column(self.loglines_tree, 'Remote Host', 1)
        self.add_column(self.loglines_tree, 'Status', 2)
        self.add_column(self.loglines_tree, 'Bytes Sent', 3)
        self.add_column(self.loglines_tree, 'Logline', 4)

        #определить прокручиваемую область для отображения файла журнала
        self.loglines_window = gtk.ScrolledWindow()

        #объединить все вместе
        self.window.add(self.outer_vbox)
        self.outer_vbox.pack_start(self.toolbar, False, False)
        self.outer_vbox.pack_start(self.file_chooser)
        self.outer_vbox.pack_start(self.loglines_window)
        self.loglines_window.add(self.loglines_tree)

        #сделать элементы видимыми
        self.window.show_all()

        #при этом элемент выбора файла должен оставаться невидимым
        self.file_chooser.hide()

    def add_column(self, tree_view, title, columnId, sortable=True):
        column = gtk.TreeViewColumn(title, gtk.CellRendererText() ,
                                    text=columnId)
        column.set_resizable(True)
        column.set_sort_column_id(columnId)
        tree_view.append_column(column)

    def show_file_chooser(self, widget, data=None):
        """делает видимым диалог выбора файла"""
        self.file_chooser.show()

    def load_logfile(self, widget, data=None):
        """загружает данные в визуальный компонент"""
        filename = widget.get_filename()
        print "FILE-->", filename
        self.file_chooser.hide()
        self.loglines_store.clear()
        logfile = open(filename, 'r')
        for i, line in enumerate(logfile):
            line_dict = dictify_logline(line)
            self.loglines_store.append([i + 1, line_dict['remote_host'],
                line_dict['status'], int(line_dict['bytes_sent']), line])
        logfile.close()

    def quit(self, widget, data=None):
        """останавливает главный цикл событий gtk"""
        gtk.main_quit()

def main(self):
    """запускает главный цикл событий gtk"""
```

```
        gtk.main()

if __name__ == "__main__":
    l = ApacheLogViewer()
    l.main()
```

В примере приложения просмотра файла журнала веб-сервера Apache главный класс приложения называется `ApacheLogViewer` и наследует класс `object`. В нашем главном объекте нет ничего особенного, он просто объединяет в себе все части графического интерфейса.

Далее в методе `__init__()` создается объект окна. В этом примере данная операция отличается от аналогичной операции в предыдущем, «простом», примере тем, что здесь указаны размеры окна. Мы сначала указываем, что окно должно иметь размеры 640×480, а затем предписываем максимизировать его. Такая двойная установка размеров была выполнена преднамеренно. 640×480 – это довольно разумные начальные размеры, поэтому это очень неплохие значения по умолчанию. Хотя размеры 640×480 достаточно хороши, но чем окно больше, тем лучше, поэтому мы максимизируем окно. Оказывается, что первоначальная установка размеров 640×480 (или любых других размеров) считается хорошей практикой. Согласно документации к PyGTK менеджер окон может не поддерживать запрос `maximize()`. Кроме того, пользователю может понадобиться снова уменьшить размеры окна после его увеличения, поэтому есть смысл задать исходные размеры окна.

После создания окна и определения его размеров мы создаем элемент `VBox`. Это область, или ящик, с «вертикальным размещением», представляющая собой контейнерный объект. В библиотеке GTK используется концепция использования областей с вертикальным (`VBox`) и горизонтальным (`HBox`) размещением визуальных компонентов (виджетов) в окне. Основное предназначение этих областей состоит в том, чтобы вы могли «наполнять» их виджетами, помещая их в начало (сверху для `VBox` и слева для `HBox`) или в конец области. Под термином «виджет» подразумеваются обычные элементы графического интерфейса, такие как кнопки или текстовые поля. При использовании этих областей вы можете расположить виджеты в окне практически любым требуемым вам способом. Поскольку области являются контейнерами, они могут вмещать другие области, поэтому вы спокойно можете вставлять одни области в другие.

После добавления области `VBox` в окно мы добавляем панель инструментов и кнопки. Сама по себе панель инструментов – это еще одна разновидность контейнеров, и она предоставляет методы для добавления в нее компонентов. Далее мы создаем пиктограммы для кнопок, сами кнопки и подключаем к кнопкам обработчики событий. Наконец, мы добавляем кнопки на панель инструментов. Для добавления виджетов на панель инструментов `Toolbar` используется метод `insert()`, играющий ту же роль, что и метод `pack_start()` области `VBox`.

Далее мы создаем виджет выбора файлов, который позволит отыскивать файлы журналов для просмотра, и связываем его с обработчиком события. В этой части нет ничего сложного, но мы вскоре еще вернемся к ней.

После создания виджета выбора файлов мы создаем компонент списка, который будет содержать строки из файла журнала. Этот компонент состоит из двух частей: объект хранения данных (с именем `ListStore`) и визуальный компонент (`TreeView`), с которым пользователь будет взаимодействовать. Компонент хранения данных создается первым, путем определения типов данных для каждой колонки. Затем мы создаем визуальный компонент и связываем с ним компонент хранения данных.

Вслед за компонентом списка создается последний контейнер – прокручиваемая область окна, после чего все виджеты объединяются вместе. Мы помещаем в созданную ранее область `VBox` панель инструментов, виджет выбора файлов и прокручиваемую область. Список, содержащий строки из файла журнала, мы добавляем в прокручиваемую область, благодаря чему при большом количестве строк мы сможем прокручивать их.

В заключение мы делаем одни виджеты видимыми, а другие – невидимыми. Главное окно делается видимым с помощью метода `show_all()`. Этот метод делает видимыми и все вложенные компоненты. Учитывая, что мы создаем приложение с графическим интерфейсом, нам необходимо, чтобы виджет выбора файлов оставался невидимым, пока пользователь не щелкнет на кнопке «open» (открыть). Поэтому этот виджет после его создания мы делаем невидимым.

Запустив это приложение, вы сможете убедиться, что оно соответствует нашим первоначальным требованиям. Мы можем выбирать и открывать нужный нам файл журнала. Каждому полю – номер строки, имя хоста, код состояния и количество переданных байтов – соответствует своя колонка в компоненте списка, поэтому мы легко можем идентифицировать данные, просто взглянув на строку. Кроме того, мы можем выполнять сортировку по любому столбцу, просто щелкнув на соответствующем заголовке.

Создание приложения для просмотра файла журнала веб-сервера Apache с помощью curses

curses – это библиотека, облегчающая создание интерактивных приложений с текстовым интерфейсом. В отличие от библиотек графического интерфейса curses не поддерживает модель обработки событий функциями обратного вызова. Вы сами отвечаете за получение ввода от пользователя и за его обработку, тогда как в GTK задача получения ввода от пользователя обрабатывается виджетами, и библиотека сама

вызывает функции-обработчики при возникновении событий. Еще одно различие между curses и библиотеками создания графического интерфейса заключается в том, что при использовании библиотек графического интерфейса вы добавляете виджеты в некоторый контейнер и позволяете библиотеке самой заниматься отображением и обновлением экрана. При использовании библиотеки curses вам обычно самим придется заниматься выводом текста на экран.

В примере 11.3 приводится еще одна версия программы просмотра файла журнала веб-сервера Apache, реализованная с использованием модуля curses, входящего в состав стандартной библиотеки языка Python.

Пример 11.3. Приложение curses для просмотра файла журнала веб-сервера Apache

```
#!/usr/bin/env python
"""

Программа просмотра файла журнала веб-сервера Apache, реализованная на основе
библиотеки curses

Порядок использования:

    curses_log_viewer.py logfile

Этой командой будет запущено интерактивное, управляемое с клавиатуры
приложение просмотра файла журнала. Ниже приводится перечень горячих клавиш
с описанием выполняемых ими функций:

u/d   - прокрутка вверх/вниз
t     - перейти в начало файла
q     - завершить работу
b/h/s - сортировать по количеству байтов/имени хоста/коду состояния
r     - восстановить первоначальный порядок сортировки

"""

import curses
from apache_log_parser_regex import dictify_logline
import sys
import operator

class CursesLogViewer(object):
    def __init__(self, logfile=None):
        self.screen = curses.initscr()
        self.curr_topline = 0
        self.logfile = logfile
        self.loglines = []

    def page_up(self):
        self.curr_topline = self.curr_topline - (2 * curses.LINES)
        if self.curr_topline < 0:
            self.curr_topline = 0
        self.draw_loglines()
```

```
    def page_down(self):
        self.draw_loglines()

    def top(self):
        self.curr_topline = 0
        self.draw_loglines()

    def sortby(self, field):
       #self.loglines = sorted(self.loglines, key=operator.itemgetter(field))
        self.loglines.sort(key=operator.itemgetter(field))
        self.top()

    def set_logfile(self, logfile):
        self.logfile = logfile
        self.load_loglines()

    def load_loglines(self):
        self.loglines = []
        logfile = open(self.logfile, 'r')
        for i, line in enumerate(logfile):
            line_dict = dictify_logline(line)
        self.loglines.append((i + 1, line_dict['remote_host'],
        line_dict['status'], int(line_dict['bytes_sent']), line.rstrip()))
        logfile.close()
        self.draw_loglines()

    def draw_loglines(self):
        self.screen.clear()
        status_col = 4
        bytes_col = 6
        remote_host_col = 16
        status_start = 0
        bytes_start = 4
        remote_host_start = 10
        line_start = 26
        logline_cols = curses.COLS-status_col-bytes_col-remote_host_col-1
        for i in range(curses.LINES):
            c = self.curr_topline
            try:
                curr_line = self.loglines[c]
            except IndexError:
                break
            self.screen.addstr(i, status_start, str(curr_line[2]))
            self.screen.addstr(i, bytes_start, str(curr_line[3]))
            self.screen.addstr(i, remote_host_start, str(curr_line[1]))
            #self.screen.addstr(i, line_start,
                               str(curr_line[4])[logline_cols])
           self.screen.addstr(i, line_start, str(curr_line[4]), logline_cols)
            self.curr_topline += 1
        self.screen.refresh()

    def main_loop(self, stdscr):
        stdscr.clear()
        self.load_loglines()
```

```
            while True:
                c = self.screen.getch()
                try:
                    c = chr(c)
                except ValueError:
                    continue
                if c == 'd':
                    self.page_down()
                elif c == 'u':
                    self.page_up()
                elif c == 't':
                    self.top()
                elif c == 'b':
                    self.sortby(3)
                elif c == 'h':
                    self.sortby(1)
                elif c == 's':
                    self.sortby(2)
                elif c == 'r':
                    self.sortby(0)
                elif c == 'q':
                    break

    if __name__ == '__main__':
        infile = sys.argv[1]
        c = CursesLogViewer(infile)
        curses.wrapper(c.main_loop)
```

В примере 11.3 мы создали единственный класс, `CursesLogViewer`, с целью организации программного кода. В конструкторе создается экран curses и инициализируется несколько переменных. Экземпляр класса `CursesLogViewer` создается в разделе «main» программы, при этом ему передается имя файла журнала, который требуется просмотреть. Мы могли бы реализовать в приложении возможность поиска и выбора файла, но для этого пришлось бы приложить больше усилий, чем в приложении PyGTK. Кроме того, поскольку приложение будет запускаться пользователем из командной оболочки, вполне естественным будет ожидать, что пользователь сначала отыщет требуемый файл в командной строке, а затем укажет его при запуске приложения. После создания экземпляра класса `CursesLogViewer` мы передаем метод `main_loop()` функции `wrapper()` из библиотеки curses. Функция `wrapper()` переводит терминал в состояние, пригодное для работы приложения на базе curses, вызывает указанную ей функцию, а затем, перед возвратом, возвращает терминал в нормальное состояние.

Метод `main_loop()` действует как элементарный цикл обработки событий. Он ожидает, пока пользователь нажмет какую-либо клавишу на клавиатуре. После этого в соответствии с введенным символом цикл переходит к соответствующему методу (или, по крайней мере, к реализации требуемого поведения). Нажатие клавиш `u` и `d` вызывает про-

крутку вверх и вниз – за счет вызова методов `page_up()` и `page_down()`, соответственно. Метод `page_down()` просто вызывает метод `draw_loglines()`, который выводит строки на терминал, начиная с текущей строки и с верхней позиции на экране. При выводе каждой строки текущей становится следующая строка. Метод `draw_loglines()` выводит столько строк, сколько поместится на экране, а при следующем вызове он вновь начнет вывод очередной текущей строки с верхней позиции на экране. Поэтому многократный вызов `draw_loglines()` создает визуальный эффект прокрутки вниз по содержимому файла журнала. Метод `page_up()` сначала назначает текущей строку, расположенную на две страницы выше, и затем производит вывод строк вызовом метода `draw_loglines()`. Это создает визуальный эффект прокрутки вверх по содержимому файла журнала. Причина, по которой в методе `page_up()` текущей назначается строка, расположенная на две страницы выше, состоит в том, что после вывода строк текущей становится строка, расположенная внизу экрана. Такой порядок был выбран для упреждения прокрутки вниз.

Следующий метод нашего класса реализует сортировку. Мы предусмотрели сортировку по имени хоста, коду состояния и количеству байтов, отправленных в ответ на запрос. Любая попытка сортировки приводит к вызову метода `sortby()`. Метод `sortby()` сортирует список строк объекта `CursesLogViewer` по указанному полю, после чего вызывает метод `top()`. Метод `top()` назначает текущей первую строку в списке и затем выводит очередную страницу строк (которая будет первой страницей).

Последний обработчик события в нашем приложении выполняет завершение работы приложения. В этом случае просто происходит прерывание «цикла обработки событий», что позволяет методу `main_loop()` вернуть управление функции `wrapper()`, которая в свою очередь производит восстановление режима работы терминала.

Несмотря на то, что по числу строк программного кода обе версии приложения вполне сопоставимы, тем не менее, для реализации приложения на базе библиотеки curses пришлось приложить больше усилий. Возможно, это обусловлено необходимостью создавать свой собственный цикл обработки событий. Или, может быть, из-за необходимости создавать некоторое подобие графических элементов. Или, возможно, из-за того, что пришлось «рисовать» текст непосредственно на экране терминала, у нас сложилось ощущение, что пришлось выполнить больше работы. Как бы то ни было, иногда знание того, как создавать приложения на базе curses, может оказаться полезным.

На рис. 11.3 показан внешний вид приложения просмотра файла журнала, в котором строки отсортированы по количеству отправленных байтов.

Одно из улучшений, которое можно было бы внести в это приложение, – это реализовать сортировку в порядке, обратном текущему. Сделать

это достаточно просто, но мы оставим реализацию этой возможности читателям. Как еще одно улучшение можно было бы организовать просмотр всей информационной части строки журнала. Это тоже не очень сложно, но реализацию этой возможности мы также оставим за читателями в качестве упражнения.

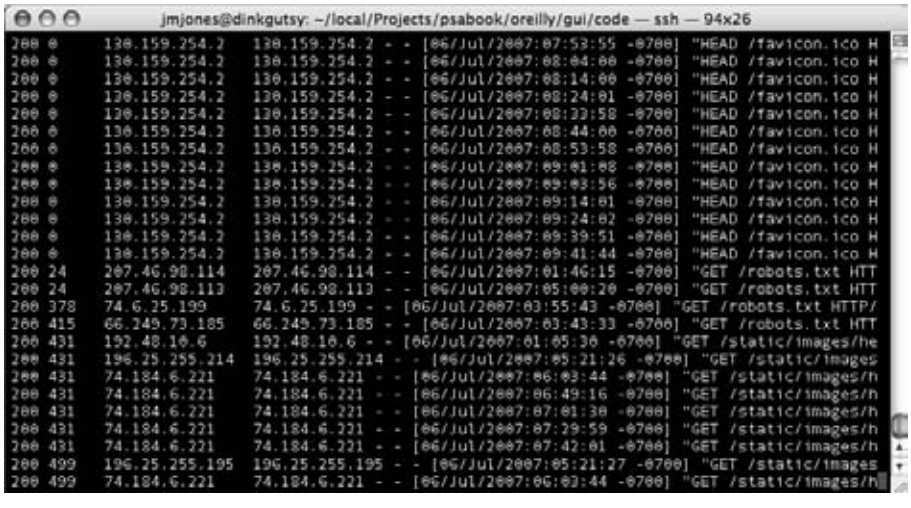


Рис. 11.3. *Содержимое файла журнала веб-сервера Apache*

Веб-приложения

Сказать, что Сеть огромна, значит преуменьшить ее истинные размеры. Сеть изобилует приложениями, которые люди используют ежедневно. Почему в Сети так много приложений? Во-первых, веб-приложения отличаются широтой доступности. Это означает, что после развертывания веб-приложения любой, кто обладает доступом к нему, может просто указать адрес URL в своем броузере и пользоваться им. Пользователям не требуется ничего загружать и устанавливать, разве только дополнения к броузеру (который сам по себе уже установлен), такие как Flash. Эта особенность особенно привлекательна для пользователей. Во-вторых, веб-приложения могут подвергаться модернизации в одностороннем порядке, причем сразу для всех пользователей. Это означает, что одна сторона (владелец приложения) может выполнить модернизацию приложения без какого-либо участия другой стороны (пользователя). Хотя в действительности это справедливо, только если вы не опираетесь на функциональные возможности, которые могут отсутствовать у пользователя. Например, если ваше модернизированное приложение опирается на новейшую версию Flash, это может потребовать от пользователей установить новую версию расшире-

ния, и все ваши преимущества могут «вылететь в трубу». Если же этого не требуется, то возможность модернизации веб-приложений становится привлекательной для обеих сторон, хотя пользователи, скорее всего, осознают это не так явно. В-третьих, броузер представляет собой в значительной степени универсальную платформу. Конечно, имеются определенные проблемы с обеспечением совместимости между броузерами, но в большинстве случаев, если вы не будете использовать специальные расширения, то веб-приложение, работающее в одном броузере и в одной операционной системе, практически наверняка будет работать в другом броузере и в другой операционной системе. Эта особенность также является привлекательной для обеих сторон. Просто со стороны разработчика придется приложить немного больше усилий, чтобы обеспечить работоспособность приложения в различных броузерах. А пользователи любят пользоваться приложениями, оставляющими за ними право выбора.

Насколько это важно для вас, как для системного администратора? Все причины, которые приводились в пользу создания приложений с графическим интерфейсом, в равной степени относятся и к веб-приложениям. Одно из преимуществ веб-приложений для системных администраторов состоит в том, что веб-приложения имеют доступ к файловой системе и таблице процессов на той машине, на которой они выполняются. Эта особенность веб-приложений делает их прекрасным решением для осуществления мониторинга системы, приложений и пользователей, а также отличным механизмом предоставления отчетов. А этот класс задач находится в области ведения системного администратора.

Хотелось бы надеяться, что вы сможете воспользоваться этими преимуществами, хотя совсем не все веб-приложения, которые вы создаете для себя или для ваших пользователей, могут давать такой эффект. Итак, какие инструменты вы можете использовать для создания веб-приложений? Поскольку эта книга о языке Python, мы, конечно же, рекомендуем использовать решения, основанные на языке Python. Но что из них выбрать? Одна из проблем состоит в том, что существует столько же платформ для разработки веб-приложений на языке Python, сколько дней в году. В настоящее время доминирующее положение занимают четыре из них – TurboGears, Django, Pylons и Zope. У каждой из этих четырех платформ имеются свои достоинства, но на наш взгляд, платформа Django лучше других соответствует теме этой книги.

Django

Django – это полнофункциональная платформа для разработки веб-приложений. Она содержит систему управления шаблонами, механизмы соединения с базами данных посредством объектно-реляционной проекции и, конечно же, сам язык Python для реализации логики приложений. Будучи «полнофункциональной» платформой, Django также

использует подход MVT (Model-View-Template – модель-представление-шаблон). Подход модель-представление-шаблон похож, если не идентичен, более общему подходу MVC (Model-View-Controller – модель-представление-контроллер). Оба способа позволяют разрабатывать веб-приложения так, чтобы не смешивать части приложений. Программный код взаимодействия с базой данных в обоих случаях представляет собой отдельную область, которая называется «моделью». Бизнес-логика выделяется в область, которая называется «представлением» в MVT и «контроллером» в MVC. А внешний интерфейс выделяется в область, которая называется «шаблоном» в MVT и «представлением» в MVC.

Приложение для просмотра файла журнала веб-сервера Apache

В следующем примере, состоящем из нескольких фрагментов программного кода, мы создадим еще одно приложение для просмотра файла журнала веб-сервера Apache, аналогичное тому, что было реализовано на базе библиотеки PyGTK. Так как мы собираемся открывать файлы журналов, чтобы позволить пользователям просматривать и сортировать их, то нам не потребуется обращаться к базе данных и наш пример будет лишен каких-либо средств подключения к базам данных. Прежде чем приступить к обсуждению примера, мы покажем вам, как создать проект и приложение в Django.

Загрузить Django можно по адресу: *http://www.djangoproject.com/*. К моменту написания этих строк последней была версия 0.96. Однако мы рекомендуем устанавливать версию из дерева разработки. После загрузки платформа устанавливается обычной командой `python setup.py install`. После установки в каталоге *site-packages* появятся дополнительные библиотеки платформы Django и сценарий `django-admin.py` в каталоге для сценариев. Обычно в UNIX-подобных системах сценарии по умолчанию устанавливаются в тот же каталог, где находится выполняемый файл `python`.

После установки Django необходимо создать проект и приложение. Проекты содержат по одному или более приложений. Кроме того, они играют роль центров конфигурации для всего веб-приложения (не путайте с приложением Django), которое вы создаете. Приложения Django – это небольшие фрагменты, которые могут многократно использоваться в разных проектах. Для нашего приложения просмотра файла журанала веб-сервера Apache мы создали проект с именем «dj_apache», выполнив команду `django-admin.py startproject dj_apache`. Эта команда создала каталог и несколько файлов. В примере 11.4 приводится дерево каталогов нового проекта.

Пример 11.4. Дерево каталогов проекта Django

```
jmjones@dinkbuntu:~/code$ tree dj_apache
```

```
dj_apache
|-- __init__.py
|-- manage.py
|-- settings.py
`-- urls.py
0 directories, 4 files
```

Теперь, когда у нас имеется проект, мы можем в рамках этого проекта создать приложение. Для этого сначала следует перейти в каталог *dj_apache*, а затем создать приложение, выполнив команду `django-admin.py startapp logview`. В результате в каталоге *dj_apache* будет создан каталог *logview* и несколько файлов. В примере 11.5 показаны все файлы и каталоги, которые теперь имеются в нашем распоряжении.

Пример 11.5. Дерево каталогов приложения Django

```
jmjones@dinkbuntu:~/tmp$ tree dj_apache/
dj_apache/
|-- __init__.py
|-- logview
|   |-- __init__.py
|   |-- models.py
|   `-- views.py
|-- manage.py
|-- settings.py
`-- urls.py
```

Как видите, каталог приложения (`logview`) содержит файлы `models.py` и `views.py`. Платформа Django следует соглашениям MVT, поэтому эти файлы помогут разделить все приложение на компоненты, соответствующие именам файлов. Файл `models.py` содержит схему базы данных, поэтому он представляет компонент Model (модель) в аббревиатуре MVT. Файл `views.py` содержит реализацию логики приложения, поэтому он представляет компонент View (представление) в этой аббревиатуре.

Здесь отсутствует компонент Template (шаблон). Компонент шаблона содержит уровень представления всего приложения. Существует несколько способов заставить платформу Django увидеть наши шаблоны. Так, в примере 11.6 показано, что мы создали подкаталог *templates* в каталоге *logview*.

Пример 11.6. Добавление каталога templates

```
jmjones@dinkbuntu:~/code$ mkdir dj_apache/logview/templates
jmjones@dinkbuntu:~/code$ tree dj_apache/
dj_apache/
|-- __init__.py
|-- logview
|   |-- __init__.py
|   |-- models.py
|   |-- templates
```

```
|   `-- views.py
|-- manage.py
|-- settings.py
`-- urls.py

2 directories, 7 files
```

Теперь мы готовы к воплощению нашего приложения. В первую очередь мы должны решить, как будут работать наши URL. Это очень простое приложение, поэтому адреса URL должны быть очень простыми. Нам потребуется выводить список файлов журналов, доступных для просмотра. Поскольку приложение обладает простой и ограниченной функциональностью, мы будем использовать адрес URL «/» для доступа к списку файлов и строку URL вида `"/viewlog/some_sort_method/some_log_file"` для указания имени файла и метода сортировки. Чтобы связать URL с определенным действием, нам необходимо обновить файл *urls.py* в корневом каталоге проекта. Содержимое файла `urls.py` для нашего проекта приводится в примере 11.7.

Пример 11.7. Конфигурация URL для проекта Django (urls.py)

```
from django.conf.urls.defaults import *

urlpatterns = patterns('',
    (r'^$', 'dj_apache.logview.views.list_files'),
    (r'^viewlog/(?P<sortmethod>.*?)/(?P<filename>.*?)/$',
      'dj_apache.logview.views.view_log'),
)
```

Файл с настройками URL достаточно прост и понятен. Этот файл опирается на использование регулярных выражений: строки URL, соответствующие регулярному выражению, отображаются на функцию представления, задаваемую явно строкой. Здесь мы отображаем URL «/» на функцию `"dj_apache.logview.views.list_files"`. Все остальные URL, соответствующие регулярному выражению `'^viewlog/(?P<sortmethod>.*?)/(?P<filename>.*?)/$'`, – на функцию `"dj_apache.logview.views.view_log"`. Когда броузер соединяется с веб-приложением Django и отсылает запрос на доступ к определенному ресурсу, Django просматривает `urls.py` в поисках элемента, регулярное выражение которого соответствует URL, и затем направляет запрос соответствующей функции.

В примере 11.8 представлен исходный текст функций представления для данного приложения, а также вспомогательной функции.

Пример 11.8. Модуль представления Django (views.py)

```
# Создайте свои представления здесь.

from django.shortcuts import render_to_response

import os
from apache_log_parser_regex import dictify_logline
import operator
```

```
log_dir = '/var/log/apache2'

def get_log_dict(logline):
    l = dictify_logline(logline)
    try:
        l['bytes_sent'] = int(l['bytes_sent'])
    except ValueError:
        bytes_sent = 0
    l['logline'] = logline
    return l

def list_files(request):
    file_list = [f for f in os.listdir(log_dir) if
                 os.path.isfile(os.path.join(log_dir, f))]
    return render_to_response('list_files.html', {'file_list': file_list})

def view_log(request, sortmethod, filename):
    logfile = open(os.path.join(log_dir, filename), 'r')
    loglines = [get_log_dict(l) for l in logfile]
    logfile.close()
    try:
        loglines.sort(key=operator.itemgetter(sortmethod))
    except KeyError:
        pass
    return render_to_response('view_logfile.html', {'loglines': loglines,
                                                    'filename': filename})
```

Функция `list_files()` получает список всех файлов, находящихся в каталоге, заданном переменной `log_dir`, и передает этот список шаблону `list_files.html`. Это все, что происходит в функции `list_files()`. Данная функция настраивается изменением значения переменной `log_dir`. Другой способ влияния на работу этой функции мог бы заключаться в хранении имени каталога журналов в базе данных. Если бы имя каталога хранилось в базе данных, мы могли бы изменять это значение без необходимости перезапускать приложение.

Функция `view_log()` принимает в качестве аргументов метод сортировки и имя файла журнала. Значения для обоих этих аргументов извлекаются из URL посредством регулярного выражения, заданного в файле *urls.py*. Для задания метода сортировки и имени файла мы использовали именованные группы в регулярном выражении в файле `urls.py`, но это не является обязательным. Аргументы, извлеченные из URL, передаются функции представления в том же порядке, в каком они были найдены в соответствующих группах. Это распространенная практика, когда в регулярном выражении разбора URL используются именованные группы, потому что благодаря такому подходу вы легко можете сказать, какие параметры извлекаются из URL, а также – как должна выглядеть строка URL.

Функция `view_log()` открывает файл журнала с именем, полученным из URL. Затем выполняется его анализ с помощью библиотеки анализа файлов журналов Apache из приведенных ранее примеров, чтобы пре-

образовать каждую строку в кортеж, в формате: статус, удаленный_хост, количество_байтов и остаток строки журнала. Затем функция view_log() сортирует список кортежей, который был получен из URL, с учетом указанного метода сортировки. В заключение функция view_log() передает полученный список шаблону view_logfile.html для отображения.

Единственное, что осталось сделать, это создать шаблоны, которые используются функциями представления для отображения информации. Шаблоны в платформе Django могут наследовать другие шаблоны, благодаря чему повышается уровень многократного использования кода и обеспечивается единообразие внешнего вида страниц. Первым мы создадим шаблон, который является родительским для двух других шаблонов. Этот шаблон будет определять общий внешний вид для других двух шаблонов в приложении. Именно поэтому мы и начнем с него. Это шаблон base.html, исходный код которого приводится в примере 11.9.

Пример 11.9. Базовый шаблон Django (base.html)

```
<html>
    <head>
        <title>
            {% block title %}Apache Logviewer - File Listing{% endblock %}
        </title>
    </head>
    <body>
        <div><a href="/">Log Directory</a></div>
        {% block content %}Empty Content Block{% endblock %}
    </body>
</html>
```

Это очень простой базовый шаблон. Возможно, это самая простая страница HTML, которую только можно получить. Единственные элементы, которые представляют здесь интерес, – это два раздела «block»: «title» и «content». Когда в родительском шаблоне определяется раздел «block», дочерний шаблон получает возможность заменить его своим собственным содержимым. Это позволяет задавать содержимое по умолчанию, которое может быть замещено в дочернем шаблоне. Блок «title» позволяет дочерним страницам определять значение, которое будет отображаться в теге <title>. Блок «content» – это типичный прием обновления «главного» раздела страницы без внесения изменений в остальную часть страницы.

В примере 11.10 приводится шаблон, с помощью которого выводится список файлов в указанном каталоге.

Пример 11.10. Шаблон Django для вывода списка файлов (list_files.html)

```
{% extends "base.html" %}

{% block title %}Apache Logviewer - File Listing{% endblock %}
```

```
{% block content %}
<ul>
{% for f in file_list %}
  <li><a href="/viewlog/linesort/{{ f }}/" >{{ f }}</a></li>
{% endfor %}
</ul>
{% endblock %}
```

На рис. 11.4 показано, как выглядит страница со списком файлов.

В этом шаблоне мы указываем, что расширяем шаблон «base.html». Это позволяет использовать все, что определено в базовом шаблоне, включать свой код в любые блоки, которые были определены, и переопределять их поведение. Именно это мы и делаем с блоками «title» и «content». В блоке «content» в цикле выполняется обход содержимого переменной `file_list`, которая была передана шаблону. Для каждого элемента в списке `file_list` создается ссылка, в результате щелчка на которой будет открыт соответствующий файл журнала.

Шаблон в примере 11.11 отвечает за создание страниц, на которые указывают ссылки из шаблона в предыдущем примере 11.10. Он отображает содержимое выбранного файла журнала.

Пример 11.11. Шаблон Django для вывода содержимого файлов (view_logfile.html)

```
{% extends "base.html" %}

{% block title %}Apache Logviewer - File Viewer{% endblock %}

{% block content %}
<table border="1">
    <tr>
        <td><a href="/viewlog/status/{{ filename }}/">Status</a></td>
        <td><a href="/viewlog/remote_host/{{ filename }}/">
            Remote Host</a></td>
```

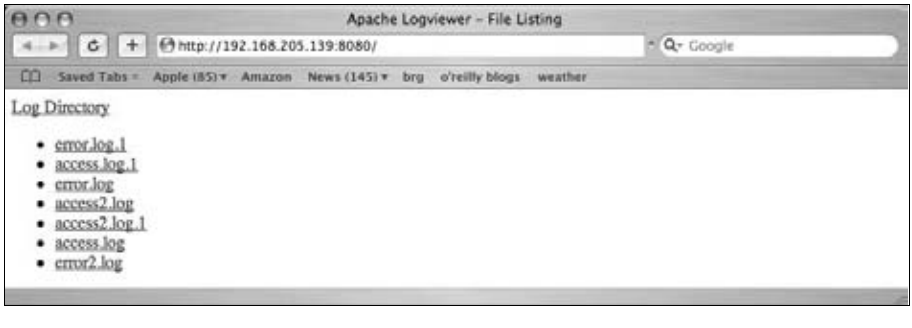


***Рис. 11.4.** Список файлов журналов веб-сервера Apache*

```
            <td><a href="/viewlog/bytes_sent/{{ filename }}/">Bytes Sent</a></td>
            <td><a href="/viewlog/linesort/{{ filename }}/">Line</a></td>
        </tr>
    {% for l in loglines %}
        <tr>
            <td>{{ l.status }}</td>
            <td>{{ l.remote_host }}</td>
            <td>{{ l.bytes_sent }}</td>
            <td><pre>{{ l.logline }}</pre></td>
        </tr>
    {% endfor %}
    </table>
    {% endblock %}
```

Шаблон в примере 11.11 наследует базовый шаблон, упоминавшийся выше, и создает таблицу в области «content». Заголовок таблицы описывает содержимое каждого столбца: код состояния, удаленный хост, количество отправленных байтов и остаток строки из файла журнала. Помимо описания содержимого, заголовки столбцов дают пользователю возможность выполнять сортировку по тому или иному столбцу. Например, если пользователь щелкнет на заголовке столбца «Bytes Sent» (передано байтов) (который является обычной ссылкой), страница будет перезагружена и программный код в сценарии представления отсортирует строки по столбцу «Bytes Sent». Щелчок на заголовке любого столбца, за исключением «Line», будет приводить к выполнению сортировки по этому столбцу в порядке возрастания. Щелчок на заголовке «Line» приведет к возврату к первоначальному порядку следования строк.

На рис. 11.5 показано, как выглядит страница приложения с оригинальным порядком следования строк, а на рис. 11.6 – как выглядит страница после выполнения сортировки по столбцу «Bytes Sent».

Это было очень простое веб-приложение, построенное на базе платформы Django. В действительности это не совсем типичное приложение. Большинство приложений Django выполняют операции с некоторыми базами данных. Данное приложение можно было бы усовершенствовать, добавив сортировку по всем столбцам в обратном порядке, фильтрацию по некоторому значению кода состояния или удаленному хосту, фильтрацию на основе критерия сравнения количества отправленных байтов с некоторым числом, возможность комбинировать различные фильтры друг с другом и дополнить их возможностями технологии AJAX. Но мы не будем выполнять все эти усовершенствования и оставим их читателю в качестве самостоятельного упражнения.

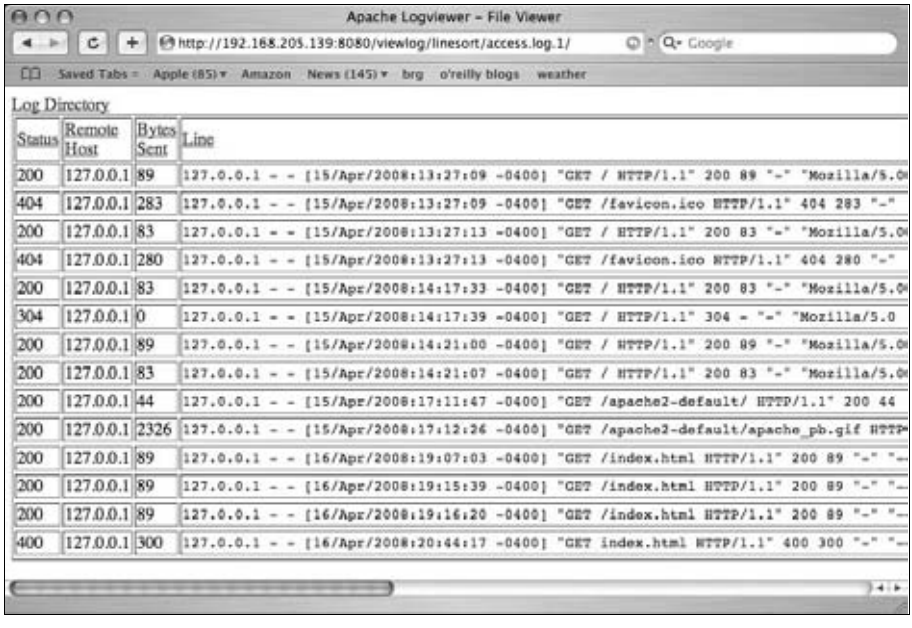


Рис. 11.5. *Просмотр содержимого файла журнала веб-сервера Apache – сортировка по номерам строк*

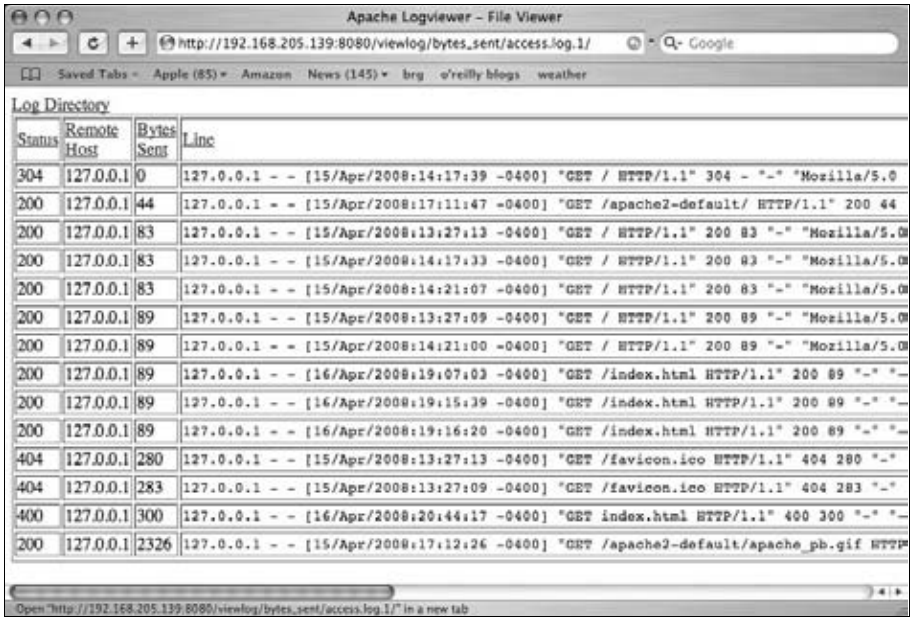


Рис. 11.6. *Просмотр содержимого файла журнала веб-сервера Apache – сортировка по количеству отправленных байтов*

Простое приложение базы данных

Выше мы уже говорили, что предыдущее приложение на платформе Django является не совсем типичным примером ее использования, так как оно не использует базу данных. Следующий пример больше соответствует типичному использованию Django, поэтому основное внимание мы сосредоточим в несколько иной области. При использовании Django для создания приложения, которое будет подключаться к базе данных, часто создаются шаблоны для отображения данных, хранящихся в базе данных, а также формы, выполняющие проверку и обработку данных, введенных пользователем. В этом примере будет показано, как создается модель базы данных с использованием объектно-реляционных проекций платформы Django, а также – как создаются шаблоны и сценарии для отображения данных, но ввод данных будет опираться на встроенный административный интерфейс платформы Django. Цель такого подхода состоит в том, чтобы показать вам, как легко и быстро можно создать интерфейс для работы с базой данных, который позволит вводить и администрировать данные.

Приложение, которое мы создадим, представляет собой приложение инвентаризации компьютерных систем. В частности, это приложение будет позволять вносить в базу данных компьютеры с их описанием, присвоенными им IP-адресами, с перечислением служб, которые выполняются на них, перечень аппаратного обеспечения, составляющего сервер, и многое другое.

Как и в предыдущем примере, мы выполним те же самые действия, чтобы создать проект и приложение Django. Ниже приводятся команды обращения к инструменту командной строки `django-admin`, создающие проект и приложение:

```
jmjones@dinkbuntu:~/code$ django-admin startproject sysmanage
jmjones@dinkbuntu:~/code$ cd sysmanage
jmjones@dinkbuntu:~/code/sysmanage$ django-admin startapp inventory
jmjones@dinkbuntu:~/code/sysmanage$
```

Эти команды создали аналогичное дерево каталогов, как и в предыдущем примере приложения для просмотра файла журнала веб-сервера Apache. Ниже приводится дерево каталогов и файлов, которые были созданы:

```
jmjones@dinkbuntu:~/code/sysmanage$ cd ../
jmjones@dinkbuntu:~/code$ tree sysmanage/
sysmanage/
|-- __init__.py
|-- inventory
|   |-- __init__.py
|   |-- models.py
|   `-- views.py
|-- manage.py
```

```
|-- settings.py
`-- urls.py
```

Создав проект и приложение, нам необходимо настроить базу данных, с которой мы будем работать. База данных SQLite предоставляет прекрасные возможности, особенно для нужд тестирования и разработки, при условии, что она не будет использоваться в рабочем окружении. Если приложение предусматривает работу более чем с одним пользователем одновременно, мы рекомендуем использовать более надежную базу данных, такую как PostgreSQL. Для настройки приложения на использование базы данных SQLite мы изменим пару строк в файле *settings.py*, расположенном в корневом каталоге проекта. Ниже показаны строки, которые мы изменили:

```
DATABASE_ENGINE = 'sqlite3'
DATABASE_NAME = os.path.join(os.path.dirname(__file__), 'dev.db')
```

В качестве механизма базы данных мы указали «sqlite3». Строка, определяющая местоположение базы данных (параметр `DATABASE_NAME`), требует дополнительных пояснений. Вместо того чтобы указать абсолютный путь к файлу базы данных, мы определили, что он всегда будет находиться в том же каталоге, что и файл *settings.py*. Атрибут `__file__` всегда хранит абсолютный путь к файлу *settings.py*. Вызов метода `os.path.dirname(__file__)` возвращает каталог, в котором находится файл *settings.py*. Полученное имя каталога и имя файла базы данных, который мы предполагаем создать, передается методу `os.path.join()`, возвращающему абсолютный путь к файлу базы данных, который зависит от каталога с приложением. Это полезный прием, который вы можете взять на вооружение при настройке параметров, связанных с местоположением файлов.

В дополнение к настройкам базы данных нам необходимо включить административный интерфейс платформы Django и наше приложение инвентаризации, наряду с другими приложениями в этом проекте. Ниже приводится соответствующий фрагмент файла *settings.py*:

```
INSTALLED_APPS = (
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.sites',
    'sysmanage.inventory',
)
```

Мы добавили в список установленных приложений `django.contrib.admin` и `sysmanage.inventory`. Это означает, что, когда мы потребуем от платформы Django создать базу данных, она создаст таблицы для всех установленных приложений.

Далее нам необходимо настроить отображение URL, так чтобы этот проект включал административный интерфейс. Ниже приводится соответствующая строка из файла настройки URL:

```
# Раскомментируйте следующую строку для включения административного интерфейса
    (r'^admin/', include('django.contrib.admin.urls')),
```

Инструмент, создавший файл *urls.py*, поместил в него строку, которая подключает административный интерфейс, но эту строку требуется раскомментировать. Как видите, чтобы подключить административный интерфейс, мы просто убрали символ #, стоявший в начале строки.

Теперь, когда мы настроили базу данных, добавили приложения административного интерфейса и инвентаризации и добавили административный интерфейс в конфигурационный файл *urls.py*, можно приступать к определению схемы базы данных. При использовании платформы Django для каждого приложения определяется своя собственная схема. В каталоге каждого приложения, в данном случае «inventory», присутствует файл с именем *models.py*, содержащий определения таблиц и столбцов, которые будут использоваться приложением. В Django, как и в других платформах разработки веб-приложений, опирающихся на использование объектно-реляционных проекций (Object-Relation Mapping, ORM), вполне возможно создавать и использовать базы данных, не написав ни одного SQL-выражения. Механизм ORM платформы Django превращает классы в таблицы, а атрибуты классов в столбцы этих таблиц. Например, следующий фрагмент программного кода является определением таблицы настроенной базы данных (этот фрагмент является частью более крупного сценария, к которому мы вскоре подойдем):

```
class HardwareComponent(models.Model):
    manufacturer = models.CharField(max_length=50)
    #в число типов входят видеокарта, сетевая карта...
    type = models.CharField(max_length=50)
    model = models.CharField(max_length=50, blank=True, null=True)
    vendor_part_number = models.CharField(max_length=50,
                                          blank=True, null=True)
    description = models.TextField(blank=True, null=True)
```

Обратите внимание, что класс `HardwareComponent` наследует класс `Model` платформы Django. Это означает, что класс `HardwareComponent` относится к типу `Model` и обладает соответствующим поведением. Каждому аппаратному компоненту мы придали несколько атрибутов: `manufacturer` (производитель), `type` (тип), `model` (модель), `vendor_part_number` (серийный номер) и `description` (описание). Реализация этих атрибутов находится в самой платформе Django. Нет, платформа не предоставляет какой-либо перечень производителей, но она реализует тип `CharField`.

Такое определение класса в приложении `inventory` создаст таблицу `inventory_hardwarecomponent` с шестью столбцами: `id`, `manufacturer`, `type`, `model`, `vendor_part_number` и `description`. Что практически в точности со-

ответствует определению класса для ORM. Фактически такое объявление *полностью* соответствует определению класса для ORM. Когда определяется класс модели, платформа Django создает соответствующую таблицу с именем, состоящим из имени приложения (все символы в нижнем регистре), за которым следуют символ подчеркивания и имя класса (все символы также в нижнем регистре). Кроме того, если не определено иное, платформа создаст в вашей таблице дополнительный столбец `id`, который будет играть роль первичного ключа. Ниже приводится код на языке SQL, создающий таблицу, которая полностью соответствует модели `HardwareComponent`:

```
CREATE TABLE "inventory_hardwarecomponent" (
    "id" integer NOT NULL PRIMARY KEY,
    "manufacturer" varchar(50) NOT NULL,
    "type" varchar(50) NOT NULL,
    "model" varchar(50) NULL,
    "vendor_part_number" varchar(50) NULL,
    "description" text NULL
)
```

Если вам когда-нибудь потребуется увидеть код на языке SQL, который платформа использует для создания базы данных, просто запустите в каталоге проекта команду `python manage.py sql myapp`, где аргумент `myapp` соответствует имени приложения.

Теперь, когда вы познакомились с ORM платформы Django, мы пройдем через создание модели базы данных для нашего приложения инвентаризации. В примере 11.12 приводится содержимое файла `model.py` для приложения inventory.

Пример 11.12. Схема базы данных (models.py)

```
from django.db import models

# Создайте здесь свои модели.

class OperatingSystem(models.Model):
    name = models.CharField(max_length=50)
    description = models.TextField(blank=True, null=True)

    def __str__(self):
        return self.name

    class Admin:
        pass

class Service(models.Model):
    name = models.CharField(max_length=50)
    description = models.TextField(blank=True, null=True)

    def __str__(self):
        return self.name

    class Admin:
        pass
```

```
class HardwareComponent(models.Model):
    manufacturer = models.CharField(max_length=50)
    #в число типов входят видеокарта, сетевая карта...
    type = models.CharField(max_length=50)
    model = models.CharField(max_length=50, blank=True, null=True)
    vendor_part_number = models.CharField(max_length=50,
                                          blank=True, null=True)
    description = models.TextField(blank=True, null=True)

    def __str__(self):
        return self.manufacturer

    class Admin:
        pass

class Server(models.Model):
    name = models.CharField(max_length=50)
    description = models.TextField(blank=True, null=True)
    os = models.ForeignKey(OperatingSystem)
    services = models.ManyToManyField(Service)
    hardware_component = models.ManyToManyField(HardwareComponent)

    def __str__(self):
        return self.name

    class Admin:
        pass

class IPAddress(models.Model):
    address = models.TextField(blank=True, null=True)
    server = models.ForeignKey(Server)

    def __str__(self):
        return self.address

    class Admin:
        pass
```

Для нашей модели мы определили пять классов: `OperatingSystem`, `Service`, `HardwareComponent`, `Server` **и** `IPAddress`. **Класс** `OperatingSystem` **позволяет нам определять различные операционные системы для серверов, которые будут учитываться приложением инвентаризации. В этом классе мы определили два атрибута:** `name` **и** `description`, **которые действительно будут необходимы нам. Можно было бы создать класс** `OperatingSystemVendor` **и определить ссылку на него в классе** `OperatingSystem`, **но в интересах сохранения простоты и понятности мы опустим упоминание о производителе операционной системы. Каждому серверу будет соответствовать единственная операционная система. Мы покажем это отношение между сервером и операционной системой, когда будем рассматривать класс** `Server`.

Класс `Service` **позволяет перечислить все службы, которые могут выполняться на сервере. В качестве примеров таких служб можно назвать веб-сервер Apache, сервер электронной почты Postfix, сервер**

DNS Bind и сервер OpenSSH. Как и класс `OperatingSystem`, этот класс имеет два атрибута: `name` и `description`. Каждый сервер может иметь множество служб. Мы покажем отношения между этими классами, когда будем рассматривать класс `Server`.

Класс `HerdwareComponent` представляет список всех аппаратных компонентов, которые могут содержаться в сервере. Этот список будет представлять интерес, только если вы сами добавляли аппаратные компоненты в приобретенную систему и в случае, если вы собирали сервер из отдельных компонентов. В классе `HardwareComponent` мы определили пять атрибутов: `manufacturer`, `type`, `model`, `vendor_part_number` и `description`. Как и в случае с изготовителем операционной системы, можно было бы создать отдельные классы для описания производителей и типов аппаратного обеспечения, а затем связать их отношениями. Но опять же, ради сохранения простоты мы предпочли не создавать такие отношения.

Класс `Server` – это основа системы инвентаризации. Каждый экземпляр класса `Server` – это отдельный сервер, информацию о котором мы собираем. Класс Server – это место, где сходятся все связи и устанавливаются отношения с тремя предыдущими классами. Прежде всего, мы дали каждому серверу атрибуты `name` и `description`. Они идентичны одноименным атрибутам в других классах. Чтобы установить отношения с другими классами, нам необходимо указать в классе `Server`, какого типа будут эти отношения. Каждый сервер будет иметь только одну операционную систему, поэтому мы создаем отношение с классом `OperatingSystem` по внешнему ключу (foreign key). Поскольку виртуализация становится все более распространенным явлением, отношение такого типа со временем потеряет свой смысл, но пока оно вполне удовлетворяет нашим потребностям. На сервере может выполняться множество служб, и служба одного и того же типа может выполняться на многих серверах, поэтому между классами `Server` и `Service` мы создали отношение типа «многие ко многим». Точно так же каждый сервер может содержать множество аппаратных компонентов, а один и тот же тип аппаратного компонента может быть установлен на множестве серверов. Поэтому классы `Server` и `HardwareComponent` мы также связали отношением типа «многие ко многим».

Наконец, класс `IPAddress` – это список всех IP-адресов всех серверов, которые должны быть учтены. Мы определили эту модель последней, чтобы подчеркнуть отношения между IP-адресами и серверами. Класс `IPAddress` имеет один атрибут и одно отношение. Атрибут `address` содержит IP-адрес в формате *XXX.XXX.XXX.XXX*. Между классами `IPAddress` и `Server` мы определили отношение по внешнему ключу, потому что один IP-адрес может принадлежать только одному серверу. Да, это выглядит слишком упрощенно, но это удовлетворяет целям демонстрации установления отношений между компонентами данных в Django.

Теперь все готово к созданию файла базы данных sqlite. Если запустить команду `python manage.py syncdb` в каталоге проекта, она создаст все от-

сутствующие таблицы для приложений, включенных в файл *settings.py*. Она также предложит создать в базе данных учетную запись суперпользователя, если предусмотрено создание таблиц аутентификации. Ниже приводится (усеченный) вывод команды `python manage.py syncdb`:

```
jmjones@dinkbuntu:~/code/sysmanage$ python manage.py syncdb
Creating table django_admin_log
Creating table auth_message
. . .
Creating many-to-many tables for Server model
Adding permission 'log entry | Can add log entry'
Adding permission 'log entry | Can change log entry'
Adding permission 'log entry | Can delete log entry'
You just installed Django's auth system, which means you don't have any
superusers defined.
Would you like to create one now? (yes/no): yes
Username (Leave blank to use 'jmjones'): E-mail address: none@none.com
Password:
Password (again): Superuser created successfully.
Adding permission 'message | Can add message'
. . .
Adding permission 'service | Can change service'
Adding permission 'service | Can delete service'
Adding permission 'server | Can add server'
Adding permission 'server | Can change server'
Adding permission 'server | Can delete server'
```

Теперь можно запустить сервер разработки Django и заняться исследованием административного интерфейса. Ниже приводится команда запуска сервера разработки Django и вывод, полученный в ходе ее выполнения:

```
jmjones@dinkbuntu:~/code/sysmanage$ python manage.py runserver 0.0.0.0:8080
Validating models...
0 errors found

Django version 0.97-pre-SVN-unknown, using settings 'sysmanage.settings'
Development server is running at http://0.0.0.0:8080/
Quit the server with CONTROL-C.
```

На рис. 11.7 показана форма регистрации. Пройдя процедуру аутентификации, мы сможем добавлять серверы, аппаратные компоненты, операционные системы и прочие данные. На рис. 11.8 показана главная страница административного интерфейса Django, а на рис. 11.9 – форма добавления нового аппаратного компонента. Полезно иметь инструмент, позволяющий сохранять и просматривать данные непротиворечивым, простым и удобным способом! Платформа Django удивительно легко справляется с реализацией простого и удобного интерфейса доступа к данным. И если это все, что вам необходимо, такой инструмент станет полезным для вас. Но это только самая верхушка возможностей платформы Django. Придумав вид отображения данных

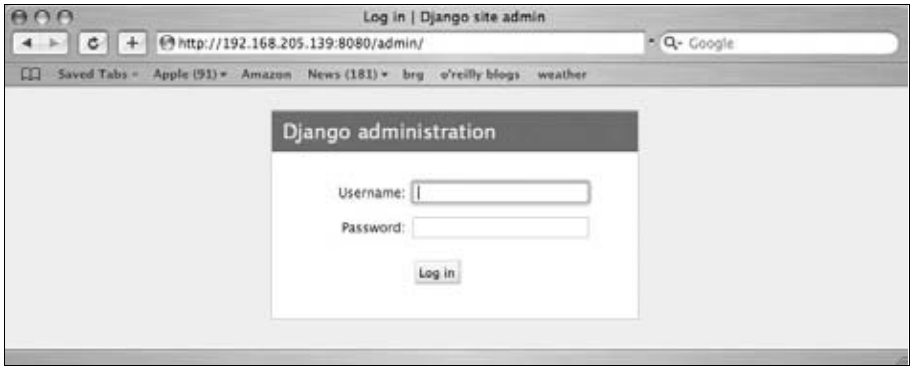


Рис. 11.7. *Страница регистрации при входе в административный интерфейс Django*

в броузере, вы наверняка сможете реализовать его с помощью Django. И, как правило, это будет не очень сложно.

Например, если бы нам потребовалось получить отдельную страницу для каждого типа операционной системы, аппаратного компонента, службы и так далее, мы могли бы это реализовать. Если бы нам потребовалось иметь возможность щелкнуть на любом из этих элементов и получить список серверов, обладающих этой характеристикой, мы также могли бы реализовать это. И если бы нам потребовалось иметь возможность щелкнуть на любом из серверов и получить подробную информацию о нем, мы смогли бы реализовать и это. Давайте теперь это сделаем. Добавим эти «потребности» к первоначальным требованиям.

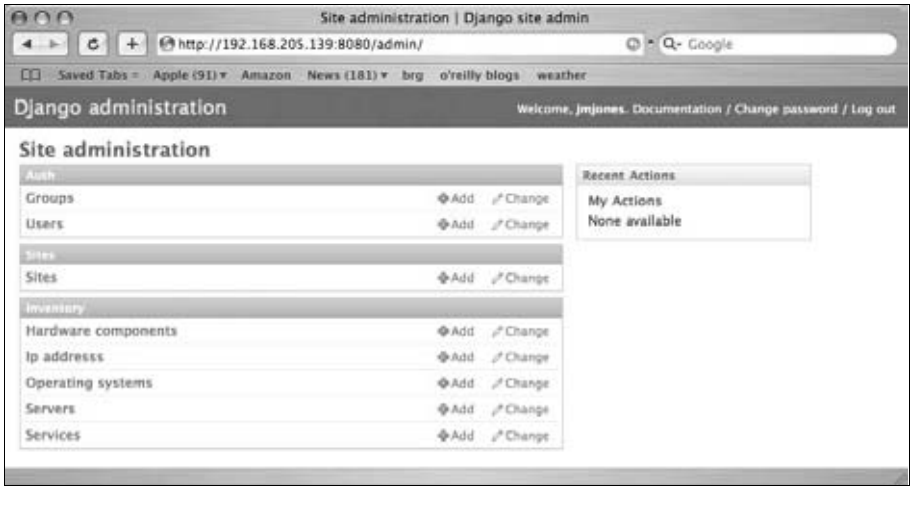


Рис. 11.8. *Главная страница административного интерфейса Django*

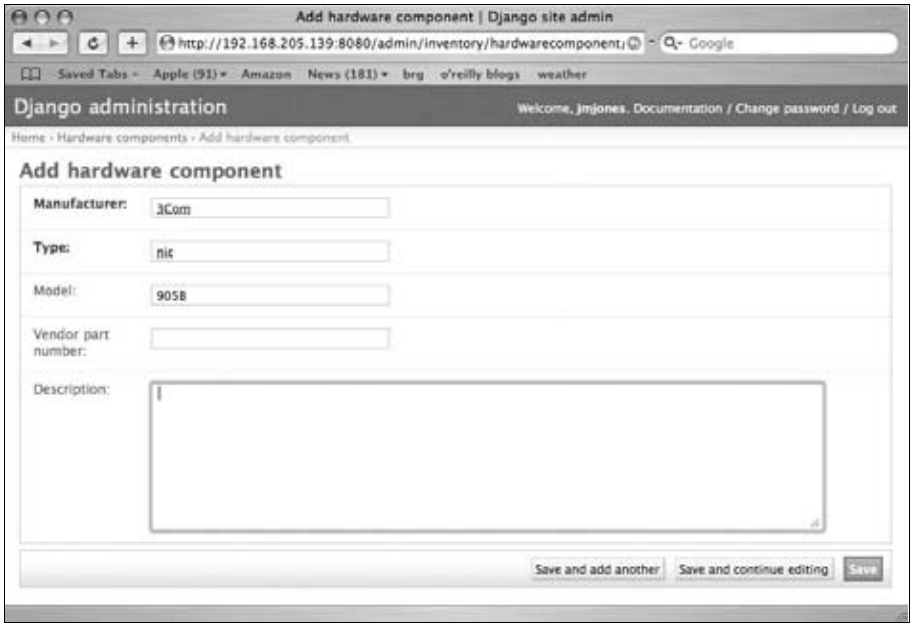


Рис. 11.9. *Форма добавления аппаратного компонента в административном интерфейсе Django*

Для начала в примере 11.13 приводится дополненный файл *urls.py*.

Пример 11.13. Отображение адресов URL (urls.py)

```
from django.conf.urls.defaults import *

urlpatterns = patterns('',
    # Пример:
    # (r'^sysmanage/', include('sysmanage.foo.urls')),

    # Раскомментируйте для включения административного интерфейса
    (r'^admin/', include('django.contrib.admin.urls')),
    (r'^$', 'sysmanage.inventory.views.main'),
    (r'^categorized/(?P<category>.*?)/(?P<category_id>.*?)/$',
            'sysmanage.inventory.views.categorized'),
    (r'^server_detail/(?P<server_id>.*?)/$',
            'sysmanage.inventory.views.server_detail'),
)
```

Мы добавили три новые строки для отображения трех адресов URL на функции. Здесь нет ничего необычного по сравнению с приложением просмотра файла журнала веб-сервера Apache. Мы отобразили адреса URL, соответствующие регулярным выражениям, на функции, при этом мы использовали в регулярных выражениях именованную группировку.

Следующее, что мы сделаем, это добавим в модуль `views` функции, которые были объявлены в файле отображения адресов URL. В примере 11.14 приводится содержимое модуля `views`.

Пример 11.14. Функции представления приложения инвентаризации (views.py)

```
# Создайте здесь свои представления.

from django.shortcuts import render_to_response
import models

def main(request):
    os_list = models.OperatingSystem.objects.all()
    svc_list = models.Service.objects.all()
    hardware_list = models.HardwareComponent.objects.all()
    return render_to_response('main.html', {'os_list': os_list,
            'svc_list': svc_list, 'hardware_list': hardware_list})

def categorized(request, category, category_id):
    category_dict = {'os': 'Operating System',
        'svc': 'Service', 'hw': 'Hardware'}
    if category == 'os':
        server_list = models.Server.objects.filter(os__exact=category_id)
        category_name = models.OperatingSystem.objects.get(id=category_id)
    elif category == 'svc':
        server_list = \
            models.Server.objects.filter(services__exact=category_id)
        category_name = models.Service.objects.get(id=category_id)
    elif category == 'hw':
        server_list = \
          models.Server.objects.filter(hardware_component__exact=category_id)
        category_name = models.HardwareComponent.objects.get(id=category_id)
    else:
        server_list = []
    return render_to_response('categorized.html', {'server_list': server_list,
        'category': category_dict[category], 'category_name': category_name})

def server_detail(request, server_id):
    server = models.Server.objects.get(id=server_id)
    return render_to_response('server_detail.html', {'server': server})
```

В файл *urls.py* мы добавили три отображения адресов URL, поэтому мы добавили три функции в файл *views.py*. Первая функция – `main()`. Она просто получает списки всех типов операционных систем, аппаратных компонентов и служб и передает их шаблону `main.html`.

В примере 11.6 мы уже создавали подкаталог *templates* в каталоге приложения. Теперь сделаем то же самое и здесь:

```
jmjones@dinkbuntu:~/code/sysmanage/inventory$ mkdir templates
jmjones@dinkbuntu:~/code/sysmanage/inventory$
```

В примере 11.15 приводится содержимое шаблона «main.html», которому функция `main()` передает данные для отображения.

Пример 11.15. Главный шаблон (main.html)

```
{% extends "base.html" %}

{% block title %}Server Inventory Category View{% endblock %}

{% block content %}
<div>
    <h2>Operating Systems</h2>
    <ul>
    {% for o in os_list %}
        <li><a href="/categorized/os/{{ o.id }}/" >{{ o.name }}</a></li>
    {% endfor %}
    </ul>
</div>
<div>
    <h2>Services</h2>
    <ul>
    {% for s in svc_list %}
        <li><a href="/categorized/svc/{{ s.id }}/" >{{ s.name }}</a></li>
    {% endfor %}
    </ul>
</div>
<div>
    <h2>Hardware Components</h2>
    <ul>
    {% for h in hardware_list %}
        <li>
            <a href="/categorized/hw/{{ h.id }}/" >{{ h.manufacturer }}</a>
        </li>
    {% endfor %}
</ul>
</div>
{% endblock %}
```

Этот шаблон не содержит ничего сложного. Он делит страницу на три части, по одной для каждой категории, которая должна отображаться. В каждой категории выводится список элементов со ссылками, щелкая на которых, можно получить список всех серверов, которые содержат указанный элемент.

Когда пользователь щелкает на одной из таких ссылок, вызывается следующая функция представления `categorized()`.

Главный шаблон передает функции представления `categorized()` категорию (`os` – в случае операционной системы, `hw` – в случае аппаратного компонента и `svc` – в случае службы) и `id` категории (то есть конкретный компонент, на котором был выполнен щелчок, например, «3Com 905b Network Card»). Функция `categorized()` принимает эти аргументы и извлекает из базы данных список всех серверов, содержащих выбранный компонент. После запроса на получения информации из базы данных функция `categorized()` передает полученные сведения шабло-

ну «categorized.html». В примере 11.16 приводится содержимое шаблона «categorized.html».

Пример 11.16. Шаблон категории (categorized.html)

```
{% extends "base.html" %}

{% block title %}Server List{% endblock %}

{% block content %}
<h1>{{ category }}::{{ category_name }}</h1>
<div>
    <ul>
    {% for s in server_list %}
        <li><a href="/server_detail/{{ s.id }}/" >{{ s.name }}</a></li>
    {% endfor %}
    </ul>
</div>
{% endblock %}
```

Шаблон «categorized.html» отображает список всех серверов, полученный от функции `categorized()`.

После этого пользователь может щелкнуть на выбранном сервере, что приведет к вызову функции представления `server_detail()`. Функция представления `server_detail()` принимает параметр с идентификатором (`id`) сервера, извлекает информацию о сервере из базы данных и передает ее шаблону «server_detail.html».

Содержимое шаблона «server_detail.html» приводится в примере 11.17. Это самый большой шаблон в приложении, но он очень простой. Его задача заключается в том, чтобы отобразить отдельные элементы данных для сервера, такие как тип операционной системы, под управлением которой работает сервер, установленные в нем аппаратные компоненты, службы, запущенные на сервере, и IP-адреса, присвоенные серверу.

Пример 11.17. Шаблон отображения информации о сервере (server_detail.html)

```
{% extends "base.html" %}

{% block title %}Server Detail{% endblock %}

{% block content %}
<div>
    Name: {{ server.name }}
</div>
<div>
    Description: {{ server.description }}
</div>
<div>
    OS: {{ server.os.name }}
</div>
<div>
    <div>Services:</div>
```

```
        <ul>
        {% for service in server.services.all %}
            <li>{{ service.name }}</li>
        {% endfor %}
        </ul>
    </div>
    <div>
        <div>Hardware:</div>
        <ul>
        {% for hw in server.hardware_component.all %}
            <li>{{ hw.manufacturer }} {{ hw.type }} {{ hw.model }}</li>
        {% endfor %}
        </ul>
    </div>
    <div>
        <div>IP Addresses:</div>
        <ul>
        {% for ip in server.ipaddress_set.all %}
            <li>{{ ip.address }}</li>
        {% endfor %}
        </ul>
    </div>
    {% endblock %}
```

Этот пример показывает, как создать довольно простое приложение базы данных, используя платформу Django. Административный интерфейс обеспечивает возможность наполнения базы данных, а добавив совсем немного строк программного кода, мы сумели создать собственные представления, позволяющие сортировать данные и перемещаться по ним, как показано на рис. 11.10, 11.11 и 11.12.

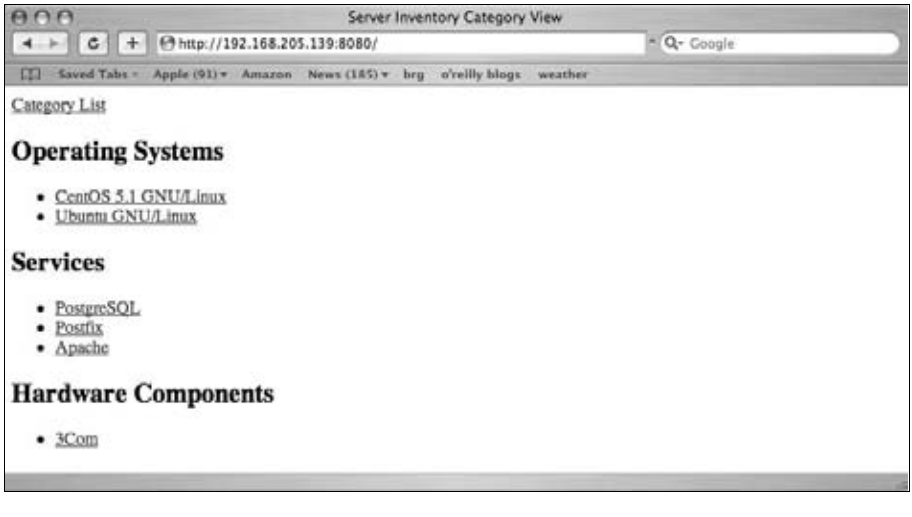


Рис. 11.10. *Основная страница приложения управления системами*

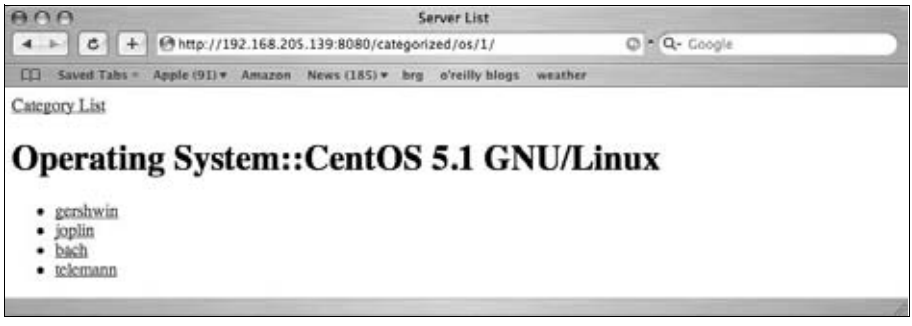


Рис. 11.11. *Приложение управления системами – категория CentOS*

Рис. 11.12. *Приложение управления системами – информация о сервере*

В заключение

Несмотря на то, что создание приложений с графическим интерфейсом, как кажется многим, не соответствует традиционным обязанностям системного администратора, тем не менее, этот навык может оказаться неоценимым. Иногда вам может даже *потребоваться* создать какое-нибудь простое приложение для одного из ваших пользователей. Иногда вам может *потребоваться* создать приложение для самого себя. Иногда вы можете склоняться к мысли, что в этом *нет необходимости*, но такие приложения могут помочь выполнить ту или иную задачу более гладко. Как только вы почувствуете, что создание приложений с графическим интерфейсом не вызывает у вас затруднений, вы будете удивлены тем, как часто вы начали их создавать.

12

Сохранность данных

Сохранность данных в простом, универсальном смысле – это сохранение данных для последующего использования. Этим подразумевается, что данные, сохраненные для последующего использования, не пропадут, если процесс, сохранивший их, завершит свою работу. Обычно сохранность данных достигается путем преобразования их в некоторый формат и запись на диск. Некоторые форматы, такие как XML или YAML, доступны человеку для чтения. Некоторые форматы, такие как файлы базы данных Berkeley DB (bdb) или SQLite, не доступны для непосредственного использования людьми.

Какие данные может потребоваться сохранять для последующего использования? Возможно, у вас имеется сценарий, который следит за датой последнего изменения файлов в каталоге, и вам необходимо периодически запускать его, чтобы узнать, какие файлы изменились с момента последнего запуска. Информация о файлах – это именно те данные, которые сохраняются для последующего использования, то есть для следующего запуска сценария. Вы могли бы сохранять эти данные в некотором файле. Представьте себе другой случай, когда у вас имеется компьютер с подозрением на проблемы, возникающие при работе с сетью, и вы решили запускать сценарий каждые 15 минут, чтобы увидеть, насколько быстро он может опросить другие компьютеры в сети. Вы могли бы сохранять время опроса в файле данных для последующего использования. В этом случае «для последующего использования» скорее относится ко времени, когда вы решите заняться исследованием этих данных, а не ко времени, когда программа, выполняющая сбор данных, обращается к ним.

Мы разобьем наше обсуждение сериализации данных на две категории: простую и реляционную.

Простая сериализация

Существует несколько способов сохранения данных на диск для последующего использования. Процесс сохранения данных на диск без сохранения отношений между частями данных мы называем «простой сериализацией». Различия между простой и реляционной сериализацией мы обсудим в разделе, описывающем реляционную сериализацию.

Pickle

Первый и, пожалуй, самый основной механизм «простой сериализации» в языке Python представлен модулем `pickle`, входящим в состав стандартной библиотеки языка. Если подумать о консервировании[1] в кулинарном смысле, идея обеспечения сохранности продуктов питания состоит в том, чтобы законсервировать их в банке для последующего использования. Кулинарная концепция прекрасно укладывается в образ действия модуля `pickle`. С помощью этого модуля вы можете записать объект на диск, завершить работу программы, вернуться позднее, снова запустить программу, прочитать объект с диска и продолжить взаимодействовать с ним.

Какими возможностями обладает модуль `pickle`? Ниже приводится список, взятый из описания модуля pickle в документации к стандартной библиотеке языка Python, где перечислены типы объектов, которые могут сохраняться с его помощью:

- None, True и False
- Целые числа, длинные целые, числа с плавающей точкой, комплексные числа
- Обычные строки и строки Юникода
- Кортежи, списки, множества и словари, содержащие только те объекты, которые могут сохраняться с помощью модуля pickle
- Функции, определенные на верхнем уровне в модуле
- Встроенные функции, определенные на верхнем уровне в модуле
- Классы, определенные на верхнем уровне в модуле
- Экземпляры классов, у которых атрибуты `__dict__` и `__setstate__()` могут сохраняться с помощью модуля pickle

Ниже показано, как выполняется сериализация объекта на диск с помощью модуля `pickle`:

```
In [1]: import pickle

In [2]: some_dict = {'a': 1, 'b': 2}

In [3]: pickle_file = open('some_dict.pkl', 'w')
```

[1] Pickle – консервировать, мариновать, солить, заквашивать. – *Прим. перев.*

```
In [4]: pickle.dump(some_dict, pickle_file)

In [5]: pickle_file.close()
```

А вот как выглядит файл с сохраненными в нем данными:

```
jmjones@dinkgutsy:~$ ls -l some_dict.pkl
-rw-r--r-- 1 jmjones jmjones 30 2008-01-20 07:13 some_dict.pkl
jmjones@dinkgutsy:~$ cat some_dict.pkl
(dp0
S'a'
p1
I1
sS'b'
p2
I2
```

Вы можете попытаться изучить формат файлов, создаваемых модулем pickle, и создавать их вручную, но мы не рекомендуем делать это.

Ниже демонстрируется, как восстановить сохраненные ранее данные:

```
In [1]: import pickle

In [2]: pickle_file = open('some_dict.pkl', 'r')

In [3]: another_name_for_some_dict = pickle.load(pickle_file)

In [4]: another_name_for_some_dict
Out[4]: {'a': 1, 'b': 2}
```

Обратите внимание, что для восстановления данных мы использовали объект, имя которого отличается от имени объекта, который сохранялся в файле. Не забывайте, что имя – это всего лишь способ сослаться на объект.

Интересно отметить, что совершенно необязательно, чтобы между файлами и сохраняемыми объектами существовало отношение «один к одному». Вы можете сохранять в одном и том же файле столько объектов, сколько места хватит на жестком диске или в файловой системе. Ниже приводится пример сохранения нескольких словарей в одном файле:

```
In [1]: list_of_dicts = [{str(i): i} for i in range(5)]

In [2]: list_of_dicts
Out[2]: [{'0': 0}, {'1': 1}, {'2': 2}, {'3': 3}, {'4': 4}]

In [3]: import pickle

In [4]: pickle_file = open('list_of_dicts.pkl', 'w')

In [5]: for d in list_of_dicts:
   ...:     pickle.dump(d, pickle_file)
   ...:
   ...:

In [6]: pickle_file.close()
```

Мы создали список словарей, объект файла, открытого в режиме для записи, затем выполнили обход списка словарей и сериализовали каждый из них в один и тот же файл. Обратите внимание, это тот же самый метод сохранения, который использовался выше для сохранения одного объекта в файл, только там мы не выполняли итерации и не вызывали метод `dump()` несколько раз.

Ниже приводится пример восстановления объектов из файла, содержащего несколько объектов, и их вывод:

```
In [1]: import pickle

In [2]: pickle_file = open('list_of_dicts.pkl', 'r')

In [3]: while 1:
   ...:     try:
   ...:         print pickle.load(pickle_file)
   ...:     except EOFError:
   ...:         print "EOF Error"
   ...:         break
   ...:
   ...:
{'0': 0}
{'1': 1}
{'2': 2}
{'3': 3}
{'4': 4}
EOF Error
```

Здесь мы создали объект файла, созданного в предыдущем примере, открытого в режиме для чтения, и повторяли попытки загружать объекты из файла, пока не было возбуждено исключение `EOFError`. Как видите, словари, полученные из файла, оказались теми же самыми (и следуют в том же порядке), что и словари, которые мы записали в файл.

Но мало того, что мы можем сохранять объекты простых встроенных типов, мы можем также сохранять объекты созданных нами типов. Ниже приводится содержимое модуля, который мы будем использовать в двух следующих примерах. Этот модуль содержит определение нашего собственного класса, экземпляры которого мы попробуем сохранить, а потом восстановить:

```
#!/usr/bin/env python

class MyClass(object):
    def __init__(self):
        self.data = []
    def __str__(self):
        return "Custom Class MyClass Data:: %s" % str(self.data)
    def add_item(self, item):
        self.data.append(item)
```

Следующий модуль импортирует модуль с нашим классом и сохраняет экземпляр этого класса в файл с помощью модуля `pickle`:

```
#!/usr/bin/env python

import pickle
import custom_class

my_obj = custom_class.MyClass()
my_obj.add_item(1)
my_obj.add_item(2)
my_obj.add_item(3)

pickle_file = open('custom_class.pkl', 'w')
pickle.dump(my_obj, pickle_file)
pickle_file.close()
```

В этом примере мы импортировали модуль с нашим классом, создали экземпляр этого класса, добавили в объект несколько элементов, затем сериализовали его. В процессе своей работы этот модуль ничего не выводит.

Далее приводится модуль, который импортирует модуль с нашим классом и затем загружает экземпляр этого класса из файла:

```
#!/usr/bin/env python

import pickle
import custom_class

pickle_file = open('custom_class.pkl', 'r')
my_obj = pickle.load(pickle_file)
print my_obj
pickle_file.close()
```

Ниже приводится вывод, полученный в ходе восстановления данных из файла:

```
jmjones@dinkgutsy:~/code$ python custom_class_unpickle.py
Custom Class MyClass Data:: [1, 2, 3]
```

Для программного кода, выполняющего восстановление данных, совершенно необязательно явно импортировать наш класс. Однако код должен иметь возможность отыскать модуль, в котором определяется наш класс. Ниже приводится модуль, который не импортирует модуль с определением класса:

```
#!/usr/bin/env python

import pickle
##import custom_class ##операция импортирования класса закомментирована

pickle_file = open('custom_class.pkl', 'r')
my_obj = pickle.load(pickle_file)
print my_obj
pickle_file.close()
```

Ниже приводится вывод, полученный в результате запуска модуля, который не импортирует класс:

```
jmjones@dinkgutsy:~/code$ python custom_class_unpickle_noimport.py
Custom Class MyClass Data:: [1, 2, 3]
```

А вот что было получено от того же самого модуля, после того как он и файл с данными были скопированы в другой каталог, где он и был запущен:

```
jmjones@dinkgutsy:~/code/cantfind$ python custom_class_unpickle_noimport.py
Traceback (most recent call last):
  File "custom_class_unpickle_noimport.py", line 7, in <module>
    my_obj = pickle.load(pickle_file)
  File "/usr/lib/python2.5/pickle.py", line 1370, in load
    return Unpickler(file).load()
  File "/usr/lib/python2.5/pickle.py", line 858, in load
    dispatch[key](self)
  File "/usr/lib/python2.5/pickle.py", line 1090, in load_global
    klass = self.find_class(module, name)
  File "/usr/lib/python2.5/pickle.py", line 1124, in find_class
    __import__(module)
ImportError: No module named custom_class
```

Последняя строка сообщает о неудачной попытке выполнить импорт, потому что модуль `pickle` не смог загрузить наш модуль с определением класса. Модуль `pickle` будет пытаться отыскать модуль, содержащий ваш класс, и импортировать его, чтобы иметь возможность вернуть объект того же типа, что и сохраненный в файле.

Все предыдущие примеры использования модуля `pickle` прекрасно работают, но существует еще один момент, о котором мы еще не упоминали. По умолчанию модуль `pickle` использует протокол сохранения `pickle.dump(object_to_pickle, pickle_file)`. Протокол – это спецификация формата записи в файл. Протокол по умолчанию использует формат, практически доступный человеку для восприятия, как было показано выше. Другая разновидность протокола – это двоичный формат. Вы можете предпочесть использовать двоичный формат, если заметите, что операция сохранения ваших объектов начинает занимать существенное время. Ниже приводится сравнение использования протокола по умолчанию и двоичного протокола:

```
In [1]: import pickle

In [2]: default_pickle_file = open('default.pkl', 'w')

In [3]: binary_pickle_file = open('binary.pkl', 'wb')

In [4]: d = {'a': 1}

In [5]: pickle.dump(d, default_pickle_file)

In [6]: pickle.dump(d, binary_pickle_file, -1)
```

```
In [7]: default_pickle_file.close()

In [8]: binary_pickle_file.close()
```

Первый файл с данными, созданный нами (с именем *default.pkl*), будет содержать данные в формате по умолчанию, практически доступном человеку для восприятия. Второй файл (с именем *binary.pkl*) будет содержать данные в двоичном формате. Обратите внимание, что мы открыли файл *default.pkl* в обычном режиме для записи (`'w'`), а файл *binary.pkl* – в режиме записи двоичных данных (`'wb'`). Единственное различие между двумя вызовами метода `dump()` заключается в том, что при сохранении в двоичном формате методу передается один дополнительный аргумент: число `-1`, означающее, что будет использоваться «высший» протокол, которым в настоящее время является двоичный протокол.

Ниже приводится шестнадцатеричный дамп двоичного файла с данными:

```
jmjones@dinkgutsy:~/code$ hexcat binary.pkl
00000000 - 80 02 7d 71 00 55 01 61 71 01 4b 01 73 2e        ..}q.U.aq.K.s.
```

А так выглядит шестнадцатеричный дамп файла с данными, сохраненными при использовании протокола по умолчанию:

```
jmjones@dinkgutsy:~/code$ hexcat default.pkl
00000000 - 28 64 70 30 0a 53 27 61 27 0a 70 31 0a 49 31 0a  (dp0.S'a'.p1.I1.
00000010 - 73 2e                                           s.
```

В этом просмотре дампа нет никакой необходимости, потому что мы можем воспользоваться простой утилитой `cat`, чтобы прочитать содержимое файла с данными, сохраненными при использовании протокола по умолчанию:

```
jmjones@dinkgutsy:~/code$ cat default.pkl
(dp0
S'a'
p1
I1
s.
```

cPickle

В стандартной библиотеке языка Python присутствует еще одна реализация библиотеки Pickle, которая стоит того, чтобы вы обратили на нее внимание. Она называется `cPickle`. Как явствует из имени, библиотека `cPickle` реализована на языке C. Ранее мы уже предлагали применять двоичный формат в случаях, когда вы начнете замечать, что на сохранение объектов требуется существенное время. В этом же случае можно попробовать использовать модуль `cPickle`. Интерфейс модуля `cPickle` в точности соответствует интерфейсу «обычного» модуля `pickle`.

shelve

Еще одну возможность сохранения данных предоставляет модуль shelve. Модуль shelve имеет простой и удобный интерфейс, упрощающий возможность сохранения множества объектов. Под этим подразумевается возможность сохранения множества объектов в одном и том же объекте-хранилище и простого их восстановления из хранилища. Сохранение объектов в хранилище shelve напоминает использование словаря в языке Python. Ниже приводится пример, в котором открывается хранилище, в него записываются данные, затем хранилище повторно открывается и из него извлекаются сохраненные данные:

```
In [1]: import shelve

In [2]: d = shelve.open('example.s')

In [3]: d
Out[3]: {}

In [4]: d['key'] = 'some value'

In [5]: d.close()

In [6]: d2 = shelve.open('example.s')

In [7]: d2
Out[7]: {'key': 'some value'}
```

Единственное отличие между использованием shelve и простого словаря состоит в том, что объект shelve создается с помощью метода shelve.open(), а не путем создания экземпляра класса dict или с помощью фигурных скобок ({}). Еще одно отличие состоит в том, что при использовании shelve по завершении работы с данными необходимо вызывать метод close() объекта shelve.

У объекта shelve имеется пара особенностей. О первой из них мы уже упоминали: по завершении работы с данными необходимо вызывать метод close(). Если этого не сделать, то любые изменения, которые были сделаны в объекте shelve, не будут сохранены. Ниже приводится пример потери данных из-за того, что объект shelve не закрывается. Для начала нам нужно создать объект shelve, сохранить в нем данные и выйти из оболочки IPython:

```
In [1]: import shelve

In [2]: d = shelve.open('lossy.s')

In [3]: d['key'] = 'this is a key that will persist'

In [4]: d
Out[4]: {'key': 'this is a key that will persist'}

In [5]: d.close()

In [6]:
Do you really want to exit ([y]/n)?
```

Теперь снова запустим IPython, откроем тот же файл хранилища, создадим в нем еще один элемент и выйдем, не закрыв объект `shelve`:

```
In [1]: import shelve

In [2]: d = shelve.open('lossy.s')

In [3]: d
Out[3]: {'key': 'this is a key that will persist'}

In [4]: d['another_key'] = 'this is an entry that will not persist'

In [5]:
Do you really want to exit ([y]/n)?
```

Теперь снова запустим оболочку IPython, откроем все тот же файл хранилища и посмотрим, что в нем имеется:

```
In [1]: import shelve

In [2]: d = shelve.open('lossy.s')

In [3]: d
Out[3]: {'key': 'this is a key that will persist'}
```

Итак, необходимо вызывать метод `close()` для всех объектов `shelve`, содержимое которых вы меняете, и которые вам хотелось бы сохранить.

Другая особенность касается изменяемых объектов. Запомните, что изменяемыми объектами называются такие объекты, значение которых можно изменять без повторного присваивания этого значения переменной. Ниже мы создаем объект `shelve`, добавляем в него элемент, который представляет собой изменяемый объект (в данном случае – список), модифицируем изменяемый объект, а затем закрываем объект `shelve`:

```
In [1]: import shelve

In [2]: d = shelve.open('mutable_lossy.s')

In [3]: d['key'] = []

In [4]: d['key'].append(1)

In [5]: d.close()

In [6]:
Do you really want to exit ([y]/n)?
```

Поскольку в этом случае вызывается метод `close()` объекта `shelve`, можно было бы ожидать, что значением ключа `'key'` будет список `[1]`. Но это не так. Ниже приводится результат попытки открыть файл хранилища, созданного выше, и прочитать из него данные:

```
In [1]: import shelve

In [2]: d = shelve.open('mutable_lossy.s')

In [3]: d
Out[3]: {'key': []}
```

В таком поведении нет ничего странного или неожиданного. В действительности эта особенность shelve описана в документации. Проблема состоит в том, что модификация сохраняемых изменяемых объектов не воспринимаются по умолчанию. Однако существует пара способов, позволяющих обойти этот недостаток. Первый из них специализированный и узконаправленный, второй – широкий и всеобъемлющий. Первый, специализированный, подход заключается в том, чтобы просто выполнить повторное присваивание по ключу в объекте shelve, как показано ниже:

```
In [1]: import shelve

In [2]: d = shelve.open('mutable_nonlossy.s')

In [3]: d['key'] = []

In [4]: temp_list = d['key']

In [5]: temp_list.append(1)

In [6]: d['key'] = temp_list

In [7]: d.close()

In [8]:
Do you really want to exit ([y]/n)?
```

При попытке восстановить сохраненный ранее объект мы получили следующее:

```
In [1]: import shelve

In [2]: d = shelve.open('mutable_nonlossy.s')

In [3]: d
Out[3]: {'key': [1]}
```

Список, к которому был добавлен элемент, сохранился.

Второй, широкий и всеобъемлющий подход заключается в изменении флага writeback объекта shelve. До сих пор мы демонстрировали вызов метода shelve.open() с единственным параметром – именем файла хранилища. Но этот метод может принимать еще и другие параметры, одним из которых является флаг writeback. Если во флаге writeback передано значение True, все записи в объекте shelve, к которым выполнялось обращение, кэшируются в памяти и затем сохраняются при вызове метода close(). Этот прием может оказаться удобным при работе с изменяемыми объектами, но за это приходится платить. Поскольку все объекты, к которым производилось обращение, кэшируются и затем сохраняются при закрытии объекта (независимо от того, изменялись они или нет), объем используемой памяти и время на запись в файл будут расти пропорционально числу объектов в хранилище, к которым производился доступ. Поэтому, если у вас имеется большое число объектов в хранилище, к которым приходится обращаться, то лучше не устанавливать флаг writeback в значение True.

В следующем примере мы устанавливаем во флаге `writeback` значение `True` и модифицируем содержимое списка, не выполняя повторное его присваивание ключу в объекте `shelve`:

```
In [1]: import shelve

In [2]: d = shelve.open('mutable_nonlossy.s', writeback=True)

In [3]: d['key'] = []

In [4]: d['key'].append(1)

In [5]: d.close()

In [6]:
Do you really want to exit ([y]/n)?
```

А теперь проверим, сохранились ли наши изменения:

```
In [1]: import shelve

In [2]: d = shelve.open('mutable_nonlossy.s')

In [3]: d
Out[3]: {'key': [1]}
```

Как мы и надеялись, изменения были сохранены.

Модуль `shelve` предлагает простой способ сохранения данных. В нем имеется пара недостатков, но в целом это очень полезный модуль.

YAML

В зависимости от того, кому задается вопрос, вы можете услышать разные толкования аббревиатуры YAML, например: «YAML ain't markup language» (YAML – это не язык разметки) или «yet another markup language» (еще один язык разметки). В любом случае – это формат данных, который часто используется для сохранения, восстановления и обновления данных в виде простого текста. Эти данные часто имеют иерархическую структуру. Самый простой, пожалуй, способ приступить к работе с YAML в языке Python состоит в том, чтобы установить с помощью утилиты `easy_install` пакет `PyYAML`. Но зачем нам использовать YAML, который еще требуется устанавливать, когда у нас имеется встроенный модуль `pickle`? Существуют две основные причины, по которым YAML оказывается предпочтительнее, чем `pickle`. Эти две причины не делают применение YAML наилучшим во всех ситуациях, но при определенных обстоятельствах они приобретают особую значимость. Во-первых, формат YAML пригоден для восприятия человеком. Его синтаксис напоминает синтаксис конфигурационных файлов. Если у вас возникают ситуации, когда необходимо предоставить возможность редактирования конфигурационных файлов, YAML будет отличным выбором. Во-вторых, синтаксические анализаторы языка YAML реализованы во многих других языках. Если вам требуется обеспечить обмен данными между приложением на языке Python и приложением,

написанном на другом языке программирования, YAML может стать неплохим решением проблемы.

После установки PyYAML вы получаете возможность сохранять и восстанавливать данные в формате YAML. Ниже приводится пример сохранения простого словаря:

```
In [1]: import yaml

In [2]: yaml_file = open('test.yaml', 'w')

In [3]: d = {'foo': 'a', 'bar': 'b', 'bam': [1, 2,3]}

In [4]: yaml.dump(d, yaml_file, default_flow_style=False)

In [5]: yaml_file.close()
```

Этот пример достаточно прост, чтобы вы могли разобраться в нем самостоятельно, и, тем не менее, мы рассмотрим его. Первое, что здесь делается, – выполняется импортирование модуля YAML (с именем yaml). Затем открывается файл в режиме для записи, который будет использоваться для сохранения данных. Далее создается словарь (с именем d), содержащий данные, которые требуется сохранить. После этого мы сохраняем словарь (с именем d) с помощью функции dump() из модуля yaml. В качестве параметров функции dump() передаются: словарь, который требуется сохранить, выходной файл и параметр, сообщающий библиотеке YAML, что запись должна производиться в блочном стиле, а не в стиле, заданном по умолчанию, который отчасти напоминает преобразование сохраняемого объекта данных в строку.

Ниже показано, как выглядит содержимое файла с данными в формате YAML:

```
jmjones@dinkgutsy:~/code$ cat test.yaml
bam:
- 1
- 2
- 3
bar: b
foo: a
```

Когда необходимо восстановить данные, мы выполняем операции, обратные тем, что выполнялись в примере с применением функции dump(). Ниже показано, как получить данные из файла YAML:

```
In [1]: import yaml

In [2]: yaml_file = open('test.yaml', 'r')

In [3]: yaml.load(yaml_file)
Out[3]: {'bam': [1, 2, 3], 'bar': 'b', 'foo': 'a'}
```

Как и в примере с функцией dump(), мы сначала импортируем модуль поддержки языка YAML (yaml). Затем создаем объект файла. На этот раз мы открываем файл на диске в режиме для чтения. Наконец вызы-

вается функция load() из модуля yaml. Функция load() возвращает словарь, эквивалентный исходному словарю.

Вы наверняка поймаете себя на том, что при использовании модуля yaml вы реализуете цикл создания данных, сохранения их на диске, затем восстановления с диска и так далее.

Возможно, вам не обязательно сохранять свои данные в формате, доступном для восприятия человеком, поэтому попробуем сохранить словарь из предыдущего примера не в блочном режиме. Ниже показано, как сохранить тот же самый словарь не в блочном режиме:

```
In [1]: import yaml

In [2]: yaml_file = open('nonblock.yaml', 'w')

In [3]: d = {'foo': 'a', 'bar': 'b', 'bam': [1, 2,3]}

In [4]: yaml.dump(d, yaml_file)

In [5]: yaml_file.close()
```

Вот как выглядит содержимое файла с данными в формате YAML:

```
jmjones@dinkgutsy:~/code$ cat nonblock.yaml
bam: [1, 2, 3]
bar: b
foo: a
```

Очень похоже на содержимое файла, записанного в блочном режиме, за исключением списка значений переменной bam. Различия между этими режимами начинают проявляться с появлением дополнительных уровней вложенности и структур данных, напоминающих массивы, таких как списки и словари. Рассмотрим пару примеров, чтобы увидеть различия. Но прежде заметим, что исследовать примеры будет проще, если отказаться от просмотра YAML-файлов с помощью утилиты cat. Аргумент с файлом в функции dump() из модуля yaml является необязательным. (Фактически в документации к PyYAML объект типа «file» называется «stream» (поток), но в действительности большой роли это не играет.) Если функция dump() не получит аргумент с файлом (или «потоком»), она выведет сериализованный объект в поток стандартного вывода. Поэтому в следующем примере мы опустили аргумент с объектом типа file и выводим результат работы функции.

Ниже сравниваются некоторые структуры данных, которые сериализуются в блочном и в не блочном режимах. В примерах, где присутствует аргумент default_flow_style, используется блочный режим форматирования, а в примерах, где аргумент default_flow_style отсутствует, используется не блочный режим форматирования:

```
In [1]: import yaml

In [2]: d = {'first': {'second': {'third': {'fourth': 'a'}}}}

In [3]: print yaml.dump(d, default_flow_style=False)
```

```
first:
  second:
    third:
      fourth: a

In [4]: print yaml.dump(d)
first:
  second:
    third: {fourth: a}

In [5]: d2 = [{'a': 'a'}, {'b': 'b'}, {'c': 'c'}]

In [6]: print yaml.dump(d2, default_flow_style=False)
- a: a
- b: b
- c: c

In [7]: print yaml.dump(d2)
- {a: a}
- {b: b}
- {c: c}

In [8]: d3 = [{'a': 'a'}, {'b': 'b'}, {'c': [1, 2, 3, 4, 5]}]

In [9]: print yaml.dump(d3, default_flow_style=False)
- a: a
- b: b
- c:
  - 1
  - 2
  - 3
  - 4
  - 5

In [10]: print yaml.dump(d3)
- {a: a}
- {b: b}
- c: [1, 2, 3, 4, 5]
```

А если нам потребуется сериализовать наш собственный класс? В этом случае модуль `yaml` ведет себя практически точно так же, как и модуль `pickle`. В следующем примере используется тот же самый модуль `custom_class`, который использовался в примере с модулем pickle.

Ниже приводится содержимое модуля, который импортирует модуль `custom_class`, создает экземпляр класса `MyClass`, добавляет несколько элементов в объект и затем сериализует его:

```
#!/usr/bin/env python

import yaml
import custom_class

my_obj = custom_class.MyClass()
my_obj.add_item(1)
my_obj.add_item(2)
```

```
my_obj.add_item(3)

yaml_file = open('custom_class.yaml', 'w')
yaml.dump(my_obj, yaml_file)
yaml_file.close()
```

Когда мы запустили этот модуль, получили следующий вывод:

```
jmjones@dinkgutsy:~/code$ python custom_class_yaml.py
jmjones@dinkgutsy:~/code$
```

То есть ничего. Это означает, что все идет так, как надо.

Ниже приводится модуль, обратный предыдущему:

```
#!/usr/bin/env python

import yaml
import custom_class

yaml_file = open('custom_class.yaml', 'r')
my_obj = yaml.load(yaml_file)
print my_obj
yaml_file.close()
```

Этот сценарий импортирует модули `yaml` и `custom_class`, создает объект файла для чтения данных из файла, созданного предыдущим сценарием, загружает объект из файла и выводит его.

Когда мы запустили этот сценарий, то получили следующее:

```
jmjones@dinkgutsy:~/code$ python custom_class_unyaml.py
Custom Class MyClass Data:: [1, 2, 3]
```

Точно такой же результат мы получили в примере, использующем модуль `pickle`, демонстрировавшемся ранее в этой главе, откуда следует, что модуль `yaml` проявляет именно такое поведение, какое мы и предполагали увидеть.

ZODB

Еще один способ сериализации данных основан на применении модуля ZODB. ZODB означает «Zope Object Database» (объектная база данных Zope). В простейших случаях использование ZODB напоминает сериализацию с помощью модуля `pickle` или `yaml`, но ZODB обладает возможностью расти вместе с вашими потребностями. Например, ZODB предоставляет механизм транзакций – на случай, если вам потребуется обеспечить атомарность своих операций. А если вам потребуется легко масштабируемое решение, вы можете использовать ZEO, систему распределенного хранения объектов.

База данных ZODB имела все шансы попасть не в раздел, описывающий «простую сериализацию», а в раздел, где рассказывается о «реляционной сериализации». Однако эта объектная база данных не совсем точно соответствует тому, что мы привыкли называть реляционными

базами данных, хотя вы без труда можете устанавливать отношения между объектами. Кроме того, мы продемонстрируем лишь некоторые из наиболее основных возможностей ZODB, поэтому в наших примерах она больше напоминает модуль `shelve`, чем реляционную базу данных. Поэтому мы и решили оставить ZODB в разделе, рассказывающем о «простой сериализации».

Установка ZODB выполняется просто – достаточно запустить команду `easy_install ZODB3`. Модуль ZODB имеет ряд зависимостей, но утилита `easy_install` благополучно разрешит их, и загрузит и установит все, что необходимо.

Для примера простейшего использования ZODB создадим объект-хранилище ZODB и добавим в него словарь и список. Ниже приводится программный код, выполняющий сериализацию словаря и списка:

```
#!/usr/bin/env python

import ZODB
import ZODB.FileStorage
import transaction

filestorage = ZODB.FileStorage.FileStorage('zodb_filestorage.db')
db = ZODB.DB(filestorage)
conn = db.open()

root = conn.root()
root['list'] = ['this', 'is', 'a', 'list']
root['dict'] = {'this': 'is', 'a': 'dictionary'}

transaction.commit()
conn.close()
```

По сравнению с pickle или YAML для инициализации работы с ZODB требуется написать на пару строк программного кода больше, но как только хранилище будет создано и инициализировано, оно используется ничуть не сложнее других альтернатив. Этот пример достаточно очевиден, особенно если учесть, что мы уже рассматривали другие примеры сохранения данных. И, тем не менее, мы быстро пройдемся по нему.

Во-первых, мы импортируем несколько модулей ZODB, а именно `ZODB`, `ZODB.FileStorage` и `transaction`. (Мы хотели бы здесь сделать небольшое замечание. Импортирование модуля, в имени которого отсутствует идентификационный префикс, выглядит несколько странно. Создается впечатление, что импортируемый модуль `transaction` должен иметь префикс `ZODB`. Но как бы то ни было, имя модуля такое, какое есть, и вам просто достаточно знать об этом. А теперь можно двигаться дальше.) Затем создается объект `FileStorage`, которому указывается имя файла, который будет использоваться как база данных. Затем создается объект `DB` и подключается к объекту `FileStorage`. Затем объект базы данных открывается с помощью метода `open()` и обретается ссылка на корневой узел объекта. С этого момента мы можем добавлять в корень

объекта свои структуры данных, что мы и делаем, используя импровизированные список и словарь. После этого мы подтверждаем изменения с помощью функции `transaction.commit()` и затем закрываем соединение с базой данных вызовом метода `conn.close()`.

Как только будет создан контейнер хранилища данных (как объект файла хранилища в этом примере) и запись данных будет подтверждена, у вас может появиться потребность восстановить эти данные. В следующем примере мы открываем ту же самую базу данных, но на этот раз мы читаем данные из файла, а не записываем в него:

```
#!/usr/bin/env python

import ZODB
import ZODB.FileStorage

filestorage = ZODB.FileStorage.FileStorage('zodb_filestorage.db')
db = ZODB.DB(filestorage)
conn = db.open()

root = conn.root()
print root.items()

conn.close()
```

И если запустить этот сценарий после того, как база данных будет наполнена, мы могли бы увидеть следующее:

```
jmjones@dinkgutsy:~/code$ python zodb_read.py
No handlers could be found for logger "ZODB.FileStorage"
[('list', ['this', 'is', 'a', 'list']), ('dict', {'this': 'is', 'a':
'dictionary'})]
```

При описании других механизмов сохранения данных мы рассматривали примеры сериализации своих собственных классов, поэтому мы покажем, как то же самое делается с помощью ZODB. Однако на этот раз мы не будем использовать тот же самый класс `MyClass` (позднее объясним, почему). Как и при использовании других механизмов, мы просто объявим свой класс, создадим экземпляр этого класса и затем передадим его механизму сериализации для сохранения на диске. Ниже приводится определение класса, который мы будем использовать в этот раз:

```
#!/usr/bin/env python

import persistent

class OutOfFunds(Exception):
    pass

class Account(persistent.Persistent):
    def __init__(self, name, starting_balance=0):
        self.name = name
        self.balance = starting_balance
    def __str__(self):
```

```
        return "Account %s, balance %s" % (self.name, self.balance)
    def __repr__(self):
        return "Account %s, balance %s" % (self.name, self.balance)
    def deposit(self, amount):
        self.balance += amount
        return self.balance
    def withdraw(self, amount):
        if amount > self.balance:
            raise OutOfFunds
        self.balance -= amount
        return self.balance
```

Это очень простой класс, имитирующий банковский счет и предназначенный для управления денежными средствами. Мы также определили исключение `OutOfFunds`, назначение которого объясним позже. Класс Account наследует класс `persistent.Persistent`. (Что касается модуля `persistent`, мы опять могли бы сделать высокопарное отступление об уместности значимого префикса в имени модуля, который предполагается использовать. Как при беглом знакомстве с этим программным кодом определить, что он использует `ZODB`? Никак. Но не будем ходить по кругу.) Наследование от класса persistent.Persistent позволяет задействовать скрытые механизмы и облегчает для ZODB сериализацию этих данных. В определении класса мы создали собственные реализации методов преобразования класса в строковую форму `__str__` и `__repr__`. Позднее вы увидите их в действии. Мы также создали методы `deposit()` и `withdraw()`. Оба метода изменяют атрибут `balance` объекта в сторону увеличения или уменьшения, в зависимости от того, какой метод вызывается. Метод `withdraw()` проверяет, достаточно ли денег на балансе (в атрибуте `balance`), прежде чем списать запрошенную сумму. Если денег недостаточно, метод `withdraw()` возбуждает исключение `OutOfFunds`, упоминавшееся выше. Оба метода, `deposit()` и `withdraw()`, возвращают остаток средств на счете после выполнения операции.

Ниже приводится программный код, который сохраняет только что описанный класс:

```
#!/usr/bin/env python

import ZODB
import ZODB.FileStorage
import transaction
import custom_class_zodb

filestorage = ZODB.FileStorage.FileStorage('zodb_filestorage.db')
db = ZODB.DB(filestorage)
conn = db.open()

root = conn.root()
noah = custom_class_zodb.Account('noah', 1000)
print noah
root['noah'] = noah
jeremy = custom_class_zodb.Account('jeremy', 1000)
```

```
print jeremy
root['jeremy'] = jeremy

transaction.commit()
conn.close()
```

Этот пример практически идентичен предыдущему примеру использования ZODB, где мы сохраняли словарь и список. Только здесь мы импортируем свой собственный модуль, создаем два экземпляра нашего класса и сохраняем эти два объекта в базе данных ZODB. Эти два объекта – счет `noah` и счет `jeremy`, каждый из которых имеет на балансе 1000 (предположим, $1000.00, но мы не идентифицировали, в какой валюте исчисляется сумма на счете).

Ниже приводится результат работы этого примера:

```
jmjones@dinkgutsy:~/code$ python zodb_custom_class.py
Account noah, balance 1000
Account jeremy, balance 1000
```

А если запустить модуль, отображающий содержимое базы данных ZODB, вот, что мы получим:

```
jmjones@dinkgutsy:~/code$ python zodb_read.py
No handlers could be found for logger "ZODB.FileStorage"
[('jeremy', Account jeremy, balance 1000), ('noah', Account noah, balance
1000)]
```

Наш пример не только создал два объекта, как ожидалось, но и сохранил их на диск для последующего использования.

А как нам открыть базу данных и изменить суммы на счетах? Все наши усилия были бы бессмысленны, не будь такой возможности. Ниже приводится фрагмент, открывающий базу данных, созданную ранее, и выполняющий перевод 300 (по-видимому, долларов) со счета `noah` на счет `jeremy`:

```
#!/usr/bin/env python

import ZODB
import ZODB.FileStorage
import transaction
import custom_class_zodb

filestorage = ZODB.FileStorage.FileStorage('zodb_filestorage.db')
db = ZODB.DB(filestorage)
conn = db.open()

root = conn.root()
noah = root['noah']
print "BEFORE WITHDRAWAL"
print "================="
print noah
jeremy = root['jeremy']
print jeremy
```

```
print "-----------------"

transaction.begin()
noah.withdraw(300)
jeremy.deposit(300)
transaction.commit()

print "AFTER WITHDRAWAL"
print "================"
print noah
print jeremy
print "----------------"

conn.close()
```

Ниже приводятся результаты работы этого сценария:

```
jmjones@dinkgutsy:~/code$ python zodb_withdraw_1.py
BEFORE WITHDRAWAL
=================
Account noah, balance 1000
Account jeremy, balance 1000
-----------------
AFTER WITHDRAWAL
================
Account noah, balance 700
Account jeremy, balance 1300
----------------
```

А если запустить наш сценарий, отображающий содержимое базы данных ZODB, то увидим, что данные сохранились:

```
jmjones@dinkgutsy:~/code$ python zodb_read.py
[('jeremy', Account jeremy, balance 1300), ('noah', Account noah, balance
700)]
```

Сумма на счете `noah` уменьшилась с 1000 до 700, а сумма на счете `jeremy` увеличилась с 1000 до 1300.

Причина, по которой мы отказались от использования класса `MyClass`, состоит в том, что нам хотелось продемонстрировать работу с транзакциями. Один из классических способов сделать это – продемонстрировать их использование при работе с банковскими счетами. Если вам требуется гарантировать благополучный перевод средств с одного счета на другой без потери средств, то транзакции будут первым инструментом, на который стоит обратить внимание. Ниже приводится пример, где используются транзакции в цикле и показано, что деньги никуда не пропадают:

```
#!/usr/bin/env python

import ZODB
import ZODB.FileStorage
import transaction
import custom_class_zodb
```

```
filestorage = ZODB.FileStorage.FileStorage('zodb_filestorage.db')
db = ZODB.DB(filestorage)
conn = db.open()

root = conn.root()
noah = root['noah']
print "BEFORE TRANSFER"
print "==============="
print noah
jeremy = root['jeremy']
print jeremy
print "-----------------"

while True:
    try:
        transaction.begin()
        jeremy.deposit(300)
        noah.withdraw(300)
        transaction.commit()
    except custom_class_zodb.OutOfFunds:
        print "OutOfFunds Error"
        print "Current account information:"
        print noah
        print jeremy
        transaction.abort()
        break

print "AFTER TRANSFER"
print "=============="
print noah
print jeremy
print "----------------"

conn.close()
```

Это некоторая модификация предыдущего примера сценария, выполняющего перевод средств. Только на этот раз вместо одного перевода он выполняет переводы по 300 единиц со счета `noah` на счет `jeremy`, пока на счету `noah` не окажется недостаточно средств для перевода. В момент, когда на счету оказывается недостаточно средств, сценарий выводит сообщение о том, что возникло исключение, и информацию о текущем состоянии счетов. После этого вызывается метод `abort()` транзакции и выполнение цикла прерывается. Кроме того, сценарий выводит информацию до и после цикла транзакций. Пока транзакции совершаются, и до и после операции общий объем средств на счетах составляет 2000, поскольку изначально на каждом счете имелась сумма 1000.

Ниже приводится результат запуска этого сценария:

```
jmjones@dinkgutsy:~/code$ python zodb_withdraw_2.py
BEFORE TRANSFER
===============
Account noah, balance 700
```

```
Account jeremy, balance 1300
-----------------
OutOfFunds Error
Current account information:
Account noah, balance 100
Account jeremy, balance 2200
AFTER TRANSFER
==============
Account noah, balance 100
Account jeremy, balance 1900
----------------
```

Перед началом цикла переводов на счете `noah` имелась сумма 700 единиц и на счете `jeremy` – 1300 единиц, итого 2000. Когда возникло исключение `OutOfFunds`, на счете `noah` имелось 100 единиц и на счете `jeremy` – 2200, итого 2300. В блоке «AFTER TRANSFER» (после перевода) на счете `noah` осталось 100 единиц и на счете `jeremy` – 1900, итого 2000. Итак, когда возникло исключение, перед тем как был вызван метод `transaction.abort()`, имелись лишние 300 единиц, появление которых невозможно было бы объяснить. Но прерывание транзакции ликвидировало эту проблему.

База данных ZODB представляет собой решение, занимающее промежуточное положение между простыми и реляционными инструментами. Она проста в использовании. Объект, сохраняемый на диске, соответствует объекту в памяти как до сохранения, так и после восстановления. Но у этого инструмента имеются такие дополнительные особенности, как транзакции. База данных ZODB стоит того, чтобы на нее обратили внимание, когда изначально требуется достаточно простой механизм отображения объектов, расширенные возможности которого могут потребоваться позже.

В заключение раздела о простой сериализации: иногда все, что вам требуется, – это просто сохранять и восстанавливать объекты Python. Все инструменты, которые мы рассмотрели здесь, прекрасно справляются с этой задачей. У каждого из них есть свои сильные и слабые стороны. Когда возникнет такая необходимость, вы сможете заняться исследованием и выяснить, какой из инструментов лучше подходит для вас и вашего проекта.

Реляционная сериализация

Иногда простой сериализации бывает недостаточно. Иногда возникает потребность в использовании мощи реляционного анализа. Под реляционной сериализацией подразумевается либо сохранение объектов Python вместе с информацией об их отношениях с другими объектами Python, либо сохранение реляционных данных (например, в реляционной базе данных) и предоставление объектного интерфейса к этим данным.

SQLite

Иногда полезно сохранять и работать с данными более структурированным способом, с учетом отношений между ними. Здесь мы будем говорить о семействе инструментов хранения информации, которые называются реляционными базами данных, или СУРБД (системы управления реляционными базами данных). Мы полагаем, что ранее вам уже приходилось использовать такие реляционные базы данных, как MySQL, PostgreSQL или Oracle. Если это так, у вас не должно возникать проблем при чтении этого раздела.

Согласно информации, что приводится на веб-сайте, SQLite – «это библиотека программного обеспечения, реализующая самодостаточный, безсерверный, не требующий настройки механизм базы данных SQL с поддержкой транзакций». Что все это означает? Этот механизм базы данных работает не в виде отдельного процесса на сервере, а в том же самом процессе, что и ваш программный код, и вы можете обращаться к нему как к библиотеке. Данные находятся в файле, а не во множестве каталогов, разбросанных по нескольким файловым системам. И вместо того, чтобы настраивать имя хоста, номер порта, имя пользователя, пароль и так далее, для организации доступа к данным вы просто указываете в своем программном коде имя файла базы данных, созданного библиотекой SQLite. Это предложение также означает, что SQLite является базой данных с достаточно широкими возможностями. Проще говоря, это предложение указывает на два главных преимущества SQLite: простота в использовании и обладание возможностями, присущими «настоящим» базам данных. Еще одно преимущество состоит в ее распространенности. Поддержка SQLite обеспечивается большинством языков программирования в большинстве основных операционных систем.

Теперь, когда вы знаете причины, которые могут побудить к использованию этой базы данных, посмотрим, как ею пользоваться. Мы взяли следующие определения таблиц из примера, где использовалась платформа Django в главе 11. Предположим, что у нас имеется файл с именем *inventory.sql*, содержащий следующий текст:

```
BEGIN;
CREATE TABLE "inventory_ipaddress" (
    "id" integer NOT NULL PRIMARY KEY,
    "address" text NULL,
    "server_id" integer NOT NULL
)
;
CREATE TABLE "inventory_hardwarecomponent" (
    "id" integer NOT NULL PRIMARY KEY,
    "manufacturer" varchar(50) NOT NULL,
    "type" varchar(50) NOT NULL,
    "model" varchar(50) NULL,
    "vendor_part_number" varchar(50) NULL,
```

```
    "description" text NULL
)
;
CREATE TABLE "inventory_operatingsystem" (
    "id" integer NOT NULL PRIMARY KEY,
    "name" varchar(50) NOT NULL,
    "description" text NULL
)
;
CREATE TABLE "inventory_service" (
    "id" integer NOT NULL PRIMARY KEY,
    "name" varchar(50) NOT NULL,
    "description" text NULL
)
;
CREATE TABLE "inventory_server" (
    "id" integer NOT NULL PRIMARY KEY,
    "name" varchar(50) NOT NULL,
    "description" text NULL,
    "os_id" integer NOT NULL REFERENCES "inventory_operatingsystem" ("id")
)
;
CREATE TABLE "inventory_server_services" (
    "id" integer NOT NULL PRIMARY KEY,
    "server_id" integer NOT NULL REFERENCES "inventory_server" ("id"),
    "service_id" integer NOT NULL REFERENCES "inventory_service" ("id"),
    UNIQUE ("server_id", "service_id")
)
;
CREATE TABLE "inventory_server_hardware_component" (
    "id" integer NOT NULL PRIMARY KEY,
    "server_id" integer NOT NULL REFERENCES "inventory_server" ("id"),
    "hardwarecomponent_id" integer
        NOT NULL REFERENCES "inventory_hardwarecomponent" ("id"),
    UNIQUE ("server_id", "hardwarecomponent_id")
)
;
COMMIT;
```

Тогда мы могли бы создать базу данных SQLite следующей командой:

```
jmjones@dinkgutsy:~/code$ sqlite3 inventory.db < inventory.sql
```

Конечно, здесь мы предполагаем, что вы уже установили SQLite. В системах Ubuntu и Debian установка выполняется простой командой `apt-get install sqlite3`. В системах Red Hat следует выполнить команду `yum install sqlite`. Для других дистрибутивов Linux, не имеющих установочных пакетов, других систем UNIX или для Windows вы можете загрузить исходные тексты и скомпилированные файлы по адресу *http://www.sqlite.org/download.html*.

Предположим, что библиотека SQLite установлена в системе и база данных была благополучно создана. Мы продолжим нашу работу с ней, начав с «подключения» к базе данных и заполнения ее некоторыми данными. Ниже показано все, что необходимо сделать для подключения к базе данных SQLite:

```
In [1]: import sqlite3

In [2]: conn = sqlite3.connect('inventory.db')
```

Все, что нам потребовалось, – это импортировать библиотеку SQLite и затем вызвать функцию `connect()` в модуле `sqlite3`. Функция `connect()` возвращает объект соединения с базой данных, которому мы присвоили имя `conn` и который мы будем использовать в оставшейся части примера. Далее с помощью объекта соединения мы выполняем запрос, добавляющий данные в базу:

```
In [3]: cursor = conn.execute("insert into inventory_operatingsystem (name,
description) values ('Linux', '2.0.34 kernel');")
```

Метод `execute()` возвращает объект курсора базы данных, поэтому мы решили дать ему имя `cursor`. Обратите внимание, что мы указали значения только для полей `name` и `description` и опустили значение для поля `id`, которое является первичным ключом. Через мгновение вы увидите, что это поле получило свое значение. Поскольку это запрос на добавление данных в базу, а не запрос на выборку, то мы не ждем от запроса результирующего набора данных; поэтому мы просто будем просматривать курсор и извлекать любые результаты, которые он может хранить:

```
In [4]: cursor.fetchall()
Out[4]: []
```

Ничего, как мы и ожидали.

```
In [5]: conn.commit()

In [6]:
```

В действительности мы не должны были подтверждать операцию добавления данных. Эти изменения все равно будут сброшены на диск, когда позднее мы закроем соединение с базой данных. Но никогда не помешает явно вызвать метод `commit()`, когда известно, что эти данные должны быть записаны.

Теперь, когда мы создали и заполнили базу данных SQLite, попробуем прочитать записанные данные обратно. Для начала запустим оболочку IPython, импортируем модуль sqlite3 и создадим соединение с файлом базы данных:

```
In [1]: import sqlite3

In [2]: conn = sqlite3.connect('inventory.db')
```

Теперь мы выполним запрос `select` и получим курсор с результатами:

```
In [3]: cursor = conn.execute('select * from inventory_operatingsystem;')
```

И, наконец, извлечем данные из курсора:

```
In [4]: cursor.fetchall()
Out[4]: [(1, u'Linux', u'2.0.34 kernel')]
```

Это те самые данные, которые были добавлены выше. Значение полей `name` и `description` хранятся в Юникоде. А поле `id` заполнено целым числом. Обычно, когда производится вставка данных в базу и при этом не указывается значение поля первичного ключа, база данных сама заполнит его, автоматически получая следующее уникальное значение для этого поля.

Теперь, когда вы познакомились с основными приемами взаимодействия с базой данных SQLite, реализация соединения таблиц, обновления данных и более сложных операций – это, в значительной степени, вопрос времени. База данных SQLite обеспечивает отличную возможность сохранения данных, особенно когда данные используются единственным сценарием или только несколькими пользователями одновременно. Говоря другими словами, SQLite прекрасно подходит для решения небольших задач. Однако интерфейс модуля sqlite3 остается слишком сложным.

Storm ORM

Несмотря на то, что простого SQL-интерфейса к базе данных вполне достаточно для извлечения, изменения, добавления и удаления данных в базе, тем не менее, часто бывает удобнее не отказываться от простоты и удобства языка Python. За последние несколько лет в способах доступа к базам данных появилось новое направление – объектно-ориентированное представление данных, хранящихся в базе. Это направление называется объектно-реляционной проекцией (Object-Relational Mapping, ORM). В терминах ORM объект на языке программирования может соответствовать одной строке в одной таблице базы данных. Таблицы, связанные отношениями внешнего ключа, могут быть доступны в виде атрибутов такого объекта.

Storm – это инструмент ORM, который недавно был выпущен как продукт, распространяемый с открытыми исходными текстами, компанией Canonical, которая ведет разработку дистрибутива Linux – Ubuntu. Storm – это относительно новый продукт среди средств доступа к базам данных для языка Python, но к нему уже проявляется пристальное внимание и мы полагаем, что он станет одним из основных средств ORM в языке Python.

Теперь мы попробуем использовать Storm для доступа к данным в базе, которая была определена в разделе «SQLite». Первое, что нам следует сделать, – это создать отображение для интересующих нас таблиц. По-

скольку мы уже обращались к таблице inventory_operatingsystem и добавили в нее одну запись, мы продолжим работу с этой таблицей. Ниже показано, как выглядит отображение при использовании библиотеки Storm:

```
import storm.locals

class OperatingSystem(object):
    __storm_table__ = 'inventory_operatingsystem'
    id = storm.locals.Int(primary=True)
    name = storm.locals.Unicode()
    description = storm.locals.Unicode()
```

Это самое обычное определение класса. Здесь нет ничего сверхъестественного. Здесь не наследуется какой-то другой класс, кроме встроенного типа object. Зато имеется несколько атрибутов. Единственное, что выглядит немного странно, – это атрибут __storm_table__. С его помощью библиотека Storm определяет, для доступа к какой таблице будет использоваться этот объект. Пока все выглядит достаточно просто и вполне обычно, и, тем не менее, во всем этом все-таки есть капелька магии. Например, атрибут name отображается на поле name в таблице inventory_operatingsystem, а атрибут description отображается на поле description в той же таблице. Как? Магия. Любой атрибут, присутствующий в классе проекции Storm, автоматически отображается на одноименное поле в таблице, имя которой определяется атрибутом __storm_table__.

А что, если нам не нужно, чтобы атрибут description объекта отображался на поле description? Тогда просто передайте методу storm.locals.Type имя требуемого поля в именованном аргументе name. Например, изменив определение атрибута description на такое: dsc = storm.locals.Unicode(name='description'), вы тем самым свяжете атрибут dsc объекта OperatingSystem с тем же самым полем (то есть с полем description). Но тогда на описание нужно будет ссылаться не как на атрибут mapped_object.description, а как на атрибут mapped_object.dsc.

Теперь, когда у нас имеется класс проекции на таблицу в базе данных, попробуем добавить в нее еще одну строку. В дополнение к нашему древнему дистрибутиву Linux на ядре 2.0.34 мы добавим Windows 3.1.1:

```
import storm.locals
import storm_model
import os

operating_system = storm_model.OperatingSystem()
operating_system.name = u'Windows'
operating_system.description = u'3.1.1'

db = storm.locals.create_database('sqlite:///%s' % os.path.join(os.getcwd(),
                                  'inventory.db'))

store = storm.locals.Store(db)
store.add(operating_system)
store.commit()
```

В этом примере мы импортировали модули `storm.locals`, `storm_model` и `os`. Затем мы создали экземпляр класса `OperatingSystem` и присвоили значения его атрибутам `name` и `description`. (Обратите внимание: в качестве значений этих атрибутов мы использовали строки Юникода.) Затем мы создали объект базы данных, вызвав функцию `create_database()`, и передали этому методу путь к файлу нашей базы данных SQLite, *inventory.db*. Вы могли бы подумать, что объект базы данных будет использоваться для добавления данных в базу, но это не так, по крайней мере, не напрямую. Сначала нам нужно создать объект `Store`, передав объект базы данных конструктору. После этого мы можем добавить объект `operating_system` в объект `store`. В заключение вызывается метод `commit()` объекта `store`, чтобы подтвердить добавление объекта `operating_system` в базу данных.

Мы также хотели бы убедиться, что вставленные данные действительно были записаны в базу данных. Поскольку это база данных SQLite, можно было бы просто воспользоваться инструментом командной строки `sqlite3`. Но если сделать это, то у нас не будет причин написать программный код, извлекающий данные из базы с помощью Storm. Итак, ниже приводится простая утилита, которая извлекает и выводит все записи из таблицы `inventory_operatingsystem` (хотя и в довольно уродливом виде):

```
import storm.locals
import storm_model
import os

db = storm.locals.create_database('sqlite:///%s' % os.path.join(os.getcwd(),
                                  'inventory.db'))

store = storm.locals.Store(db)

for o in store.find(storm_model.OperatingSystem):
    print o.id, o.name, o.description
```

Первые несколько строк в этом примере поразительно напоминают первые несколько строк предыдущего примера. Отчасти это сходство обусловлено тем, что мы просто скопировали программный код из одного файла в другой. Впрочем, это не главное. Основная же причина заключается в том, что в обоих случаях необходимо выполнить одни и те же подготовительные действия, прежде чем сценарии смогут «общаться» с базой данных. Здесь используются те же инструкции импортирования, что и в предыдущем примере. У нас имеется объект `db`, который возвращает функция `create_database()`. У нас имеется объект `store`, созданный конструктором `Store()`, которому был передан объект `db`. Но теперь вместо добавления объекта в хранилище (в объект `store`) мы вызываем метод `find()` объекта `store`. Этот конкретный вызов метода `find()` (то есть `store.find(storm_model.OperatingSystem)`) возвращает множество всех объектов `storm_model.OperatingSystem`. Поскольку класс `OperatingSystem` является проекцией на таблицу `inventory_operatingsystem`,

Storm отыщет все подходящие записи в таблице `inventory_operatingsystem` и создаст объект `OperatingSystem` для каждой из них. Для каждого объекта `OperatingSystem` выводятся значения атрибутов `id`, `name` и `description`. Эти атрибуты являются проекциями на одноименные поля записей в базе данных.

В нашей базе данных уже имеется одна запись, добавленная в более раннем примере, приводившемся в разделе «SQLite». Давайте посмотрим, что получится, если запустить этот сценарий. Мы могли бы ожидать, что будет выведена одна запись, хотя она и была добавлена без использования библиотеки Storm:

```
jmjones@dinkgutsy:~/code$ python storm_retrieve_os.py
1 Linux 2.0.34 kernel
```

Это в точности соответствует нашим ожиданиям. Теперь сначала попробуем запустить сценарий, добавляющий новую запись, а затем снова запустим сценарий, извлекающий данные. На этот раз он должен вывести старую запись, добавленную ранее (система на базе ядра Linux 2.0.34), и только что добавленную запись (Windows 3.1.1):

```
jmjones@dinkgutsy:~/code$ python storm_add_os.py
jmjones@dinkgutsy:~/code$ python storm_retrieve_os.py
1 Linux 2.0.34 kernel
2 Windows 3.1.1
```

И снова мы получили именно то, что и ожидали получить.

Но что, если нам потребуется фильтровать данные? Предположим, что нам потребуется увидеть только те операционные системы, название которых начинается с последовательности символов «Lin». Ниже приводится фрагмент программного кода, который делает именно это:

```
import storm.locals
import storm_model
import os

db = storm.locals.create_database('sqlite:///%s' % os.path.join(os.getcwd(),
                                  'inventory.db'))

store = storm.locals.Store(db)

for o in store.find(storm_model.OperatingSystem,
                    storm_model.OperatingSystem.name.like(u'Lin%')):
    print o.id, o.name, o.description
```

Этот пример идентичен предыдущему примеру, где использовался метод `store.find()`, за исключением того, что в этом примере методу `store.find()` передается второй параметр: критерий поиска. Вызов `Store.find(storm_model.OperatingSystem, storm_model.OperatingSystem.name.like(u'Lin%'))` сообщает библиотеке Storm, что требуется отыскать все объекты `OperatingSystem`, у которых значение атрибута `name` начинается со строки Юникода `Lin`. Каждое значение в наборе результатов выводится точно так же, как и в предыдущем примере.

И когда мы запустим этот фрагмент, мы увидим следующее:

```
jmjones@dinkgutsy:~/code$ python storm_retrieve_os_filter.py
1 Linux 2.0.34 kernel
```

В базе данных по-прежнему присутствует запись с названием операционной системы «Windows 3.1.1», но она была отфильтрована, потому что не начинается со строки «Lin».

SQLAlchemy ORM

В то время как библиотека Storm только начинает обретать сторонников и находится на стадии формирования сообщества, библиотека SQLAlchemy уже является доминирующим средством ORM для языка Python. Своим подходом к решению проблемы она напоминает Storm. Вероятно, лучше было бы сказать, что «библиотека Storm своим подходом к решению проблемы напоминает SQLAlchemy», поскольку библиотека SQLAlchemy появилась раньше. Но, как бы то ни было, для демонстрации SQLAlchemy мы воспользуемся все той же таблицей `inventory_operatingsystem`, для работы с которой только что использовали библиотеку Storm.

Ниже приводится определение таблицы и объекта для отображения таблицы `inventory_operatingsystem`:

```
#!/usr/bin/env python

import os
from sqlalchemy import create_engine
from sqlalchemy import Table, Column, Integer, Text, VARCHAR, MetaData
from sqlalchemy.orm import mapper
from sqlalchemy.orm import sessionmaker

engine = create_engine('sqlite:///%s' % os.path.join(os.getcwd(),
                       'inventory.db'))

metadata = MetaData()
os_table = Table('inventory_operatingsystem', metadata,
    Column('id', Integer, primary_key=True),
    Column('name', VARCHAR(50)),
    Column('description', Text()),
)

class OperatingSystem(object):
    def __init__(self, name, description):
        self.name = name
        self.description = description

    def __repr__(self):
        return "<OperatingSystem('%s','%s')>" % (self.name, self.description)

mapper(OperatingSystem, os_table)
Session = sessionmaker(bind=engine, autoflush=True, transactional=True)
session = Session()
```

Самое существенное различие между примерами использования Storm и SQLAlchemy заключается в определении таблицы, которое используется библиотекой SQLAlchemy для создания проекции вместе с классом таблицы.

Теперь, когда у нас имеется определение таблицы, можно написать программный код, выполняющий запрос всех записей из таблицы:

```
#!/usr/bin/env python

from sqlalchemy_inventory_definition import session, OperatingSystem

for os in session.query(OperatingSystem):
    print os
```

Если запустить этот фрагмент теперь, когда в таблице уже имеются некоторые данные, записанные туда в предыдущих примерах, мы увидим следующее:

```
$ python sqlalchemy_inventory_query_all.py <OperatingSystem('Linux','2.0.34
kernel')>
<OperatingSystem('Windows','3.1.1')>
</OperatingSystem></OperatingSystem>
```

Если бы нам потребовалось создать еще одну запись, мы легко могли бы сделать это, просто создав объект `OperatingSystem` и добавив его в объект `session`:

```
#!/usr/bin/env python

from sqlalchemy_inventory_definition import session, OperatingSystem

ubuntu_710 = OperatingSystem(name='Linux', description='2.6.22-14 kernel')
session.save(ubuntu_710)
session.commit()
```

В результате в таблицу будет добавлена другая запись с операционной системой Linux на другом ядре, более современном. Запустив сценарий, запрашивающий все записи, еще раз, мы получим:

```
$ python sqlalchemy_inventory_query_all.py
<OperatingSystem('Linux','2.0.34 kernel')>
<OperatingSystem('Windows','3.1.1')>
<OperatingSystem('Linux','2.6.22-14 kernel')>
```

Фильтрация результатов в SQLAlchemy выполняется также просто. Например, если бы нам потребовалось выбрать все объекты `OperatingSystem`, в которых значение атрибута `name` начинается с последовательности символов «Lin», мы могли бы написать следующий сценарий:

```
#!/usr/bin/env python

from sqlalchemy_inventory_definition import session, OperatingSystem

for os in session.query(OperatingSystem).filter(
                            OperatingSystem.name.like('Lin%')):
    print os
```

И мы могли бы получить следующие результаты:

```
$ python sqlalchemy_inventory_query_filter.py
<OperatingSystem('Linux','2.0.34 kernel')>
<OperatingSystem('Linux','2.6.22-14 kernel')>
```

Это был лишь краткий обзор возможностей библиотеки SQLAlchemy. За дополнительной информацией об использовании SQLAlchemy обращайтесь на веб-сайт *http://www.sqlalchemy.org/*. Или приобретите книгу Рика Коупленда (Rick Coupland) «Essential SQLAlchemy» (O'Reilly).

ПОРТРЕТ ЗНАМЕНИТОСТИ: SQLACLCHEMY

Майк Байер (Mike Bayer)

Майкл Байер – подрядчик на поставку программного обеспечения из Нью-Йорка, обладающий десятилетним опытом работы с реляционными базами данных всех форм и размеров. После создания множества собственных библиотек абстракции доступа к базам данных на таких языках программирования, как C, Java и Perl, и после нескольких лет практической работы с огромными, состоящими из нескольких серверов, системами Oracle для высшей лиги по бейсболу он написал SQLAlchemy, как «основной инструмент» для создания кода SQL и для работы с базами данных. Цель его состоит в том, чтобы способствовать появлению инструмента мирового класса для языка Python, помогающего превратить Python в широко популярную платформу программирования, каковой он достоин быть.

В заключение

В этой главе мы рассмотрели несколько различных инструментов, позволяющих сохранять данные для последующего использования. Иногда вам будет требоваться нечто простое и легковесное, как модуль `pickle`. Иногда вам будет требоваться нечто более полнофункциональное, как SQLAlchemy ORM. Как уже было показано, при использовании языка Python в вашем распоряжении имеется множество решений, от очень простых до мощных и сложных.

13

Командная строка

Введение

Командная строка имеет особое значение для системного администратора. Ни один другой инструмент не обладает таким уровнем значимости или авторитета, как командная строка. Полное овладение искусством командной строки – это своего рода обряд посвящения для большинства системных администраторов. Многие системные администраторы не считают администраторами тех, кто использует программы с графическим интерфейсом, и называют графический интерфейс костылями. Возможно, это не совсем справедливо, но это устоявшееся мнение об истинном искусстве владения профессией системного администратора.

В течение очень долгого времени системы UNIX придерживались философии, что интерфейс командной строки (Command Line Inteface, CLI) превосходит по своим возможностям любой графический интерфейс, который когда-либо разрабатывался. В свете последних событий создается впечатление, что даже Microsoft решила вернуться к своим корням. Джеффри Сновер (Jeffrey Snover), архитектор Windows Powershell, заявил: «Это было ошибкой – думать, что графический интерфейс когда-либо сможет или должен вытеснить интерфейс командной строки».

Даже создатели системы Windows, в которой в течение десятилетий имелся самый худший интерфейс командной строки из всех современных операционных систем, начинают понимать значимость интерфейса командной строки, что привело к реализации Windows Powershell. В этой книге мы не будем касаться операционной системы Windows, но это очень интересный факт, подчеркивающий важность освоения командной строки и действительную необходимость создания инструментов командной строки.

Однако недостаточно просто овладеть существующими в системе UNIX инструментами командной строки. Чтобы стать настоящим профессионалом командной строки, необходимо научиться создавать собственные инструменты и это может быть самой основной причиной, по которой вы взяли эту книгу в руки. Эта глава вас не разочарует. Закончив ее чтение, вы станете мастером по созданию инструментов командной строки на языке Python.

Это было преднамеренное решение – сконцентрировать внимание на создании инструментов командной строки в последней главе. Мы хотели сначала продемонстрировать вам широчайший выбор приемов программирования на языке Python, а в заключение рассказать вам, как можно использовать все эти навыки при создании настоящих шедевров командной строки.

Основы использования потока стандартного ввода

Самый простой путь к созданию инструментов командной строки опирается на знание того факта, что модуль `sys` позволяет обрабатывать аргументы командной строки посредством атрибута `sys.argv`. В примере 13.1 демонстрируется самый простой инструмент командной строки, какой только возможен:

Пример 13.1. sysargv.py

```
#!/usr/bin/env python

import sys
print sys.argv
```

Эти две строки программного кода возвращают на стандартный вывод все, что вводится в командной строке:

```
./sysargv.py

['./sysargv.py']
```

и

```
./sysargv.py foo
```

вернет на стандартный вывод

```
['./sysargv.py', 'test']
```

и

```
./sysargv.py foo bad for you
```

вернет на стандартный вывод

```
['./sysargv.py', 'foo', 'bad', 'for', 'you']
```

Добавим немного конкретики и слегка изменим программный код так, чтобы он подсчитывал количество аргументов командной строки, как показано в примере 13.2.

Пример 13.2. sysargv.py

```
#!/usr/bin/env python
import sys

#Индексы в языке Python начинаются с нуля, поэтому нужно исключить
#из подсчета саму команду - sys.argv[0]

num_arguments = len(sys.argv) - 1
print sys.argv, "You typed in ", num_arguments, "arguments"
```

Вы могли бы подумать: «Как все просто, теперь мне осталось лишь получить параметры командной строки по их индексам из списка `sys.argv` и написать некоторую логику их обработки». В общем вы правы, это довольно просто реализовать. Давайте добавим некоторые особенности к нашему приложению командной строки. Последнее, что мы могли бы сделать, – это вывести сообщение об ошибке в поток стандартного вывода сообщений об ошибках в случае отсутствия аргументов командной строки, как показано в примере 13.3.

Пример 13.3. sysargv-step2.py

```
#!/usr/bin/env python
import sys

num_arguments = len(sys.argv) - 1

#В случае отсутствия аргументов вывести сообщение
#в поток стандартного вывода сообщений об ошибках.
if num_arguments == 0:
    sys.stderr.write('Hey, type in an option silly\n')
else:
    print sys.argv, "You typed in ", num_arguments, "arguments"
```

Использование `sys.argv` при создании инструментов командной строки зачастую является неправильным выбором, несмотря на всю его простоту. В стандартной библиотеке языка Python имеется модуль `optparse`, который берет на себя решение одной из самых неудобных задач в создании качественного инструмента командной строки. Даже для самых крошечных «разовых» инструментов лучше использовать `optparse`, чем `sys.argv`, так как у «разовых» инструментов есть свойство со временем превращаться в нормальные рабочие инструменты. В следующем разделе мы объясним, почему лучше использовать модуль `optparse`, но суть ответа заключается в том, что наличие хорошего модуля разбора аргументов позволит разрешить самые необычные ситуации.

Введение в optparse

Как упоминалось в предыдущем разделе, даже самый маленький сценарий может воспользоваться преимуществами модуля optparse при выполнении разбора параметров командной строки. Приступая к изучению возможностей optparse, интереснее будет начать со своеобразного примера «Hello World», который выполняет обработку параметров и аргументов. В примере 13.4 приводится наш сценарий «Hello World».

Пример 13.4. Hello World для модуля optparse

```
#!/usr/bin/env python
import optparse

def main():
    p = optparse.OptionParser()
    p.add_option('--sysadmin', '-s', default="BOFH")
    options, arguments = p.parse_args()
    print 'Hello, %s' % options.sysadmin

if __name__ == '__main__':
    main()
```

Запуская этот сценарий, мы можем получить различные варианты вывода:

```
$ python hello_world_optparse.py
Hello, BOFH

$ python hello_world_optparse.py --sysadmin Noah
Hello, Noah

$ python hello_world_optparse.py -s Jeremy
Hello, Jeremy

$ python hello_world_optparse.py --infinity Noah
Usage: hello_world_optparse.py [options]

hello_world_optparse.py: error: no such option: --infinity
```

В нашем маленьком примере мы видели, что можно использовать как короткий -s, так и длинный --sysadmin параметры, а также значение по умолчанию. Наконец, мы увидели встроенную возможность обработки ошибок, когда указали неверный параметр и убедились в удобочитаемости, которой так не хватает языку Perl.

Простые шаблоны использования optparse

Шаблон использования без ключей

В предыдущем разделе мы упоминали, что модуль optparse может с успехом использоваться даже в маленьких сценариях. В примере 13.5 показан простой шаблон использования модуля optparse, где сценарий

не предусматривает наличие ключей командной строки, что не мешает ему использовать преимущества optparse.

Пример 13.5. Клон команды ls

```
#!/usr/bin/env python
import optparse
import os

def main():
    p = optparse.OptionParser(description="Python 'ls' command clone",
                              prog="pyls",
                              version="0.1a",
                              usage="%prog [directory]")
    options, arguments = p.parse_args()
    if len(arguments) == 1:
        path = arguments[0]
        for filename in os.listdir(path):
            print filename
    else:
        p.print_help()

if __name__ == '__main__':
    main()
```

В этом примере мы реализовали на языке Python свою версию команды ls, которая принимает единственный аргумент – путь к каталогу, содержимое которого требуется вывести. Мы не предусматриваем даже наличие дополнительных ключей, но по-прежнему можем пользоваться возможностями модуля optparse, опираясь на них при выборе пути выполнения программы. Сначала при создании экземпляра класса OptionParser мы предоставляем некоторую информацию о реализации и добавляем инструкции о порядке использования для потенциальных пользователей инструмента. Затем мы проверяем количество аргументов, и если их число больше или меньше одного, мы выводим инструкцию о порядке использования инструмента с помощью метода p.print_help(). Ниже приводится пример правильного использования нашего инструмента, которому передается имя текущего каталога «.»:

```
$ python no_options.py .
.svn
hello_world_optparse.py
no_options.py
```

А теперь посмотрим, что произойдет, если запустить сценарий без аргументов:

```
$ python no_options.py
Usage: pyls [directory]

Python 'ls' command clone

Options:
```

```
--version   show program's version number and exit
-h, --help  show this help message and exit
```

Интересно, что поведение, связанное с вызовом метода `p.print_help()`, которое мы определили для случая запуска сценария, когда число аргументов не равно точно одному, равносильно запуску сценария с ключом `--help`:

```
$ python no_options.py --help
Usage: pyls [directory]

Python 'ls' command clone

Options:
--version   show program's version number and exit
-h, --help  show this help message and exit
```

А так как мы определили параметр `--version`, то при его использовании мы получим следующее:

```
$ python no_options.py --version
0.1a
```

В этом примере модуль `optparse` оказался полезен даже при создании «разового» сценария, который может не получить дальнейшего развития.

Шаблон true/false

Бывает очень удобно иметь возможность установить некоторый признак в значение true или false. Классическим примером такого шаблона могут служить ключи: `--quiet`, который подавляет вывод в поток стандартного вывода, и `--verbose`, при установке которого программа переходит в режим вывода более подробной информации. В примере 13.6 показано, как это может выглядеть:

Пример 13.6. Увеличение и уменьшение подробности вывода

```
#!/usr/bin/env python
import optparse
import os

def main():
    p = optparse.OptionParser(description="Python 'ls' command clone",
                              prog="pyls",
                              version="0.1a",
                              usage="%prog [directory]")
    p.add_option("--verbose", "-v", action="store_true",
                 help="Enables Verbose Output",default=False)
    options, arguments = p.parse_args()
    if len(arguments) == 1:
        if options.verbose:
            print "Verbose Mode Enabled"
        path = arguments[0]
```

```
            for filename in os.listdir(path):
                if options.verbose:
                    print "Filename: %s " % filename
                elif options.quiet:
                    pass
                else:
                    print filename
        else:
            p.print_help()

if __name__ == '__main__':
    main()
```

Используя ключ `--verbose`, мы тем самым повышаем уровень подробности информации, выводимой в поток стандартного вывода. Посмотрим, какая информация выводится на каждом из уровней подробности. Сначала нормальный уровень:

```
$python true_false.py /tmp
.aksusb
alm.log
amt.log
authTokenData
FLEXnet
helloworld
hsperfdata_ngift
ics10003
ics12158
ics13342
icssuis501
MobileSync.lock.f9e26440fe5adbb6bc42d7bf8f87c1e5fc61a7fe
summary.txt
```

Теперь в режиме подробного вывода:

```
$ python true_false.py --verbose /tmp
Verbose Mode Enabled
Filename: .aksusb
Filename: alm.log
Filename: amt.log
Filename: authTokenData
Filename: FLEXnet
Filename: helloworld
Filename: hsperfdata_ngift
Filename: ics10003
Filename: ics12158
Filename: ics13342
Filename: icssuis501
Filename: MobileSync.lock.f9e26440fe5adbb6bc42d7bf8f87c1e5fc61a7fe
Filename: summary.txt
```

Когда мы указываем ключ `--verbose`, атрибут `options.verbose` получает значение `true`, в результате чего выполняется условие и вызывается

инструкция, которая выводит «Filename:» перед фактическим именем файла. Обратите внимание, что в этом сценарии при вызове метода `p.add_option()` мы определили параметры `default=False` и `action="store_true"`, указав тем самым, что по умолчанию этот параметр будет иметь значение `false`, но если при вызове сценария будет указан ключ `--verbose`, этот параметр приобретет значение `true`. В этом заключается сущность использования логических параметров с модулем `optparse`.

Шаблон подсчета числа параметров

Если при использовании типичного инструмента командной строки операционной системы UNIX, например `tcpdump`, указать параметр `-vvv`, вы получите намного более подробный вывод, чем при использовании параметра `-vv` или `-v`. Вы можете реализовать аналогичное поведение, воспользовавшись такой возможностью модуля `optparse`, как подсчет количества одинаковых параметров. Например, если вам потребуется снабдить свой сценарий аналогичными уровнями подробности вывода, вы могли бы сделать это, как показано в примере 13.7.

Пример 13.7. Шаблон с подсчетом упоминаний параметра

```
#!/usr/bin/env python
import optparse
import os

def main():
    p = optparse.OptionParser(description="Python 'ls' command clone",
                              prog="pyls",
                              version="0.1a",
                              usage="%prog [directory]")
    p.add_option("-v", action="count", dest="verbose")
    options, arguments = p.parse_args()
    if len(arguments) == 1:
        if options.verbose:
            print "Verbose Mode Enabled at Level: %s" % options.verbose
        path = arguments[0]
        for filename in os.listdir(path):
            if options.verbose == 1:
                print "Filename: %s " % filename
            elif options.verbose ==2 :
                fullpath = os.path.join(path,filename)
                print "Filename: %s | Byte Size: %s" % (filename,
                       os.path.getsize(fullpath))
            else:
                print filename
    else:
        p.print_help()

if __name__ == '__main__':
    main()
```

При использовании шаблона проектирования с автоматическим подсчетом упоминаний параметра мы можем на основе единственного параметра реализовать три варианта действий. Когда этот сценарий вызывается с ключом `-v`, атрибут `options.verbose` получает значение 1; когда сценарий вызывается с ключом `-vv`, атрибут `options.verbose` получает значение 2. Наш сценарий при вызове без ключей просто выводит имена файлов, при вызове с ключом `-v` он выводит слово «Filename:» перед каждым именем файла и, наконец, когда сценарий вызывается с ключом `-vv`, он выводит не только имя файла, но и его размер в байтах. Ниже показан результат вызова сценария с ключом `-vv`:

```
$ python verbosity_levels_count.py -vv /tmp
Verbose Mode Enabled at Level: 2
Filename: .aksusb | Byte Size: 0
Filename: alm.log | Byte Size: 1403
Filename: amt.log | Byte Size: 3038
Filename: authTokenData | Byte Size: 32
Filename: FLEXnet | Byte Size: 170
Filename: helloworld | Byte Size: 170
Filename: hsperfdata_ngift | Byte Size: 102
Filename: ics10003 | Byte Size: 0
Filename: ics12158 | Byte Size: 0
Filename: ics13342 | Byte Size: 0
Filename: ics14183 | Byte Size: 0
Filename: icssuis501 | Byte Size: 0
Filename: MobileSync.lock.f9e26440fe5adbb6bc42d7bf8f87c1e5fc61a7fe | Byte
Size: 0
Filename: summary.txt | Byte Size: 382
```

Шаблон с вариантами значений параметра

Иногда бывает необходимо предоставить несколько возможных значений параметра. В нашем последнем примере мы создали параметры `--verbose` и `--quiet`, но точно так же мы могли бы реализовать их как возможные варианты значений параметра `--chatty`. В примере 13.8 показано, как выглядит версия предыдущего примера, переделанная для использования вариантов значений.

Пример 13.8. Шаблон с вариантами значений параметра

```
#!/usr/bin/env python
import optparse
import os

def main():
    p = optparse.OptionParser(description="Python 'ls' command clone",
                              prog="pyls",
                              version="0.1a",
                              usage="%prog [directory]")
    p.add_option("--chatty", "-c", action="store", type="choice",
                 dest="chatty",
```

```
                choices=["normal", "verbose", "quiet"],
                default="normal")
    options, arguments = p.parse_args()
    print options
    if len(arguments) == 1:
        if options.chatty == "verbose":
            print "Verbose Mode Enabled"
        path = arguments[0]
        for filename in os.listdir(path):
            if options.chatty == "verbose":
                print "Filename: %s " % filename
            elif options.chatty == "quiet":
                pass
            else:
                print filename
    else:
        p.print_help()

if __name__ == '__main__':
    main()
```

Если запустить эту команду с параметром без значения, как это делалось в предыдущем примере, будет получено следующее сообщение об ошибке:

```
$ python choices.py --chatty
Usage: pyls [directory]

pyls: error: --chatty option requires an argument
```

Если указать в параметре ошибочный аргумент, будет получено другое сообщение об ошибке, где будут указаны допустимые значения:

```
$ python choices.py --chatty=nuclear /tmp
Usage: pyls [directory]

pyls: error: option --chatty: invalid choice: 'nuclear' (choose from 'normal',
  'verbose', 'quiet')
```

Одно из удобств использования вариантов значений состоит в том, что в этом случае сценарий не полагается на то, что подойдет любое введенное пользователем значение аргумента. Пользователю позволяется выбирать только из тех значений, которые вы определите. Ниже показано, как выполняется команда при запуске с допустимым значением параметра:

```
$ python choices.py --chatty=verbose /tmp
{'chatty': 'verbose'}
Verbose Mode Enabled
Filename: .aksusb
Filename: alm.log
Filename: amt.log
Filename: authTokenData
Filename: FLEXnet
```

```
Filename: helloworld
Filename: hsperfdata_ngift
Filename: ics10003
Filename: ics12158
Filename: ics13342
Filename: ics14183
Filename: icssuis501
Filename: MobileSync.lock.f9e26440fe5adbb6bc42d7bf8f87c1e5fc61a7fe
Filename: summary.txt
```

Если вы обратили внимание, в первой строке вывода указано слово «chatty» как ключ, а слово «verbose» как значение. В примере выше мы добавили инструкцию `print` для вывода атрибута `options`, чтобы показать вам, как он выглядит с точки зрения программы. В заключение ниже приводится пример запуска программы со значение `quiet` в параметре `--chatty`:

```
$ python choices.py --chatty=quiet /tmp
{'chatty': 'quiet'}
```

Шаблон использования параметров с несколькими аргументами

По умолчанию для каждого параметра модуль `optparse` принимает только один аргумент, но существует возможность определить любое число аргументов. В примере 13.9 приводится сценарий, представляющий собой еще одну версию команды `ls`, который может выводить содержимое сразу двух каталогов.

Пример 13.9. Вывод содержимого двух каталогов

```
#!/usr/bin/env python
import optparse
import os

def main():
    p = optparse.OptionParser(description="Lists contents of two directories",
                              prog="pymultils",
                              version="0.1a",
                              usage="%prog [--dir dir1 dir2]")
    p.add_option("--dir", action="store", dest="dir", nargs=2)
    options, arguments = p.parse_args()
    if options.dir:
        for dir in options.dir:
            print "Listing of %s:\n" % dir
            for filename in os.listdir(dir):
                print filename
    else:
        p.print_help()

if __name__ == '__main__':
    main()
```

Если попробовать запустить этот сценарий с единственным аргументом параметра `--dir`, будет получено следующее сообщение об ошибке:

```
[ngift@Macintosh-8][H:10238][J:0]# python multiple_option_args.py --dir /tmp ↴
Usage: pymultils [--dir dir1 dir2]

pymultils: error: --dir option requires 2 arguments
```

Указав требуемое число аргументов параметра `--dir`, мы получили следующее:

```
pymultils: error: --dir option requires 2 arguments
[ngift@Macintosh-8][H:10239][J:0]# python multiple_option_args.py --dir /tmp
  /Users/ngift/Music
Listing of /tmp:

.aksusb
FLEXnet
helloworld
hsperfdata_ngift
ics10003
ics12158
ics13342
ics14183
ics15392
icssuis501
MobileSync.lock.f9e26440fe5adbb6bc42d7bf8f87c1e5fc61a7fe
summary.txt
Listing of /Users/ngift/Music:

.DS_Store
.localized
iTunes
```

Внедрение команд оболочки в инструменты командной строки на языке Python

В главе 10 мы познакомились со множеством способов использования модуля `subprocess`. Создание новых инструментов командной строки путем обертывания существующих команд сценариями на языке Python и изменения их API, или путем включения одной или нескольких команд UNIX в сценарии на языке Python представляет собой весьма интересную область для исследований. Нет ничего сложного в том, чтобы обернуть инструмент командной строки сценарием на языке Python и изменить ее поведение так, чтобы оно полнее отвечало нашим требованиям. Можно было бы, например, интегрировать в сценарий конфигурационный файл, в котором определить значения аргументов для некоторых параметров или задавать в сценарии значения по умолчанию для других аргументов. Какие бы требования не предъявлялись, с помощью модулей `subprocess` и `optparse` можно без особых проблем изменить поведение инструментов командной строки UNIX.

С другой стороны, смешивание инструментов командной строки со сценарием на языке Python позволяет создавать интересные инструменты, которые не так-то просто написать на языке C или Bash. Что вы скажете насчет смешивания команды `dd` с многопоточным сценарием, где используются очереди, `tcpdump` с библиотекой регулярных выражений для языка Python или использования специализированной версии `rsync`? Все эти «смешанные» команды UNIX 2.0 очень напоминают особенности, присутствующие в Web 2.0. Смешивание Python с утилитами UNIX приводит к появлению новых идей и к решению проблем различными способами. В этом разделе мы исследуем некоторые из этих приемов.

Шаблон проектирования «кудзу»: обертывание инструментов сценариями на языке Python

Иногда используемый инструмент командной строки не совсем точно соответствует тому, что требуется вам. Он может требовать слишком большого числа параметров или порядок следования аргументов отличается от того, к которому вы привыкли. Используя язык программирования Python, можно очень легко изменить поведение утилиты и заставить ее делать именно то, что вам необходимо. Мы назвали это шаблоном проектирования «Кудзу». Для тех, кто не знает, поясним, что кудзу – это быстрорастущее вьющееся растение, завезенное на юг Соединенных Штатов из Японии. Кудзу часто поглощает естественный ландшафт, совершенно меняя его внешний вид. С помощью языка Python вы можете делать то же самое со средой UNIX.

В следующем примере мы обернули команду `snmpdf` сценарием на языке Python, чтобы упростить ее использование. Для начала посмотрим, как выглядит обычный запуск команды:

```
[ngift@Macintosh-8][H:10285][J:0]# snmpdf -c public -v 2c example.com
Description               size (kB)           Used      Available  Used%
Memory Buffers              2067636         249560        1818076    12%
Real Memory                 2067636        1990704          76932    96%
Swap Space                  1012084             64        1012020     0%
/                          74594112       17420740       57173372    23%
/sys                             0              0              0     0%
/boot                       101086          20041          81045    19%
```

Для тех, кто не знаком с командой `snmpdf`, поясним, что она предназначена для удаленного выполнения в системах, обладающих поддержкой SNMP и настроенных для получения информации из раздела MIB, имеющего отношение к дискам. Часто инструменты командной строки, использующие протокол SNMP, обладают большим числом параметров, что осложняет их использование. Справедливости ради следует заметить, что создатели вынуждены разрабатывать инструменты, которые могли бы работать с версиями 1, 2 и 3 протокола SNMP, и дополнительно разрешать целый ворох других проблем. А что, если эти

проблемы к вам не относятся и к тому же вы достаточно ленивы? Вы можете создать собственную кудзу-версию утилиты `snmpdf`, которая принимает в качестве аргумента только имя машины. Без всяких сомнений, это возможно. В примере 13.10 показано, как могла бы выглядеть такая утилита.

Часто, чтобы изменить поведение утилиты UNIX с помощью языка Python, приходится писать больше строк программного кода, чем на языке Bash. Но, несмотря на это, мы отдаем предпочтение языку Python, потому что он позволяет использовать более богатый набор средств для расширения инструментов под свои нужды. Кроме того, вы можете протестировать этот программный код точно так же, как вы тестируете другие свои сценарии, поэтому часто дополнительный программный код – это правильный выбор в долгосрочной перспективе.

Пример 13.10. Обертка для команды snmpdf на языке Python

```
#!/usr/bin/env python
import optparse
from subprocess import call

def main():
    p = optparse.OptionParser(description="Python wrapped snmpdf command",
                              prog="pysnmpdf",
                              version="0.1a",
                              usage="%prog machine")
    p.add_option("-c", "--community", help="snmp community string")
    p.add_option("-V", "--Version", help="snmp version to use")
    p.set_defaults(community="public",Version="2c")
    options, arguments = p.parse_args()
    SNMPDF = "snmpdf"
    if len(arguments) == 1:
        machine = arguments[0]
        #Измененное действие команды snmpdf
        call([SNMPDF, "-c", options.community, "-v",options.Version, machine])
    else:
        p.print_help()

if __name__ == '__main__':
    main()
```

Этот сценарий уместился примерно в двадцать строк программного кода, но он делает нашу жизнь намного проще. Использование волшебных особенностей модуля `optparse` помогло предусмотреть значения по умолчанию для некоторых аргументов, наилучшим образом соответствующие нашим потребностям. Например, мы определили, что по умолчанию будет использоваться версия 2 протокола SNMP, так как мы знаем, что в нашем вычислительном центре используется только эта версия протокола. Кроме того, для параметра `community` мы выбрали в качестве значения по умолчанию строку `"public"`, потому что

именно это значение определено в нашей лаборатории исследований и разработки, например. Самое замечательное, что использование модуля `optparse` позволило нам гибко изменять значения параметров, не изменяя сам сценарий.

Обратите внимание, что значения по умолчанию устанавливаются с помощью метода `set_default()`, который позволяет одним вызовом устанавливать сразу все значения по умолчанию аргументов инструмента командной строки. Мы включили старые параметры, такие как `-c`, и с помощью модуля `optparse` обернули их новыми значениями, в данном случае – `options.community`. Хотелось бы надеяться, что этот пример достаточно наглядно демонстрирует, как прием «кудзу» и широкие возможности языка Python позволяют обернуть инструмент и изменить его так, чтобы он полнее отвечал нашим потребностям.

Шаблон проектирования «гибрид кудзу»: обертывание инструментов сценариями на языке Python с изменением их поведения

В последнем примере мы существенно облегчили использование утилиты `snmpdf`, но не изменили поведение инструмента. Оба инструмента выводят совершенно идентичную информацию. Другой прием, который можно использовать, позволяет не только обернуть утилиту UNIX, но и изменить ее поведение с помощью языка Python.

В следующем примере мы воспользуемся генераторами языка Python и приемами функционального программирования, чтобы отфильтровать результаты нашей команды `snmpdf` в поисках критически важной информации и затем добавить к ней флаг `"CRITICAL"`. В примере 13.11 показано, как могла бы выглядеть такая утилита.

Пример 13.11. Измененная версия команды snmpdf с применением генераторов

```
#!/usr/bin/env python
import optparse
from subprocess import Popen, PIPE
import re

def main():
    p = optparse.OptionParser(description="Python wrapped snmpdf command",
                              prog="pysnmpdf",
                              version="0.1a",
                              usage="%prog machine")
    p.add_option("-c", "--community", help="snmp community string")
    p.add_option("-V", "--Version", help="snmp version to use")
    p.set_defaults(community="public",Version="2c")
    options, arguments = p.parse_args()
    SNMPDF = "snmpdf"
    if len(arguments) == 1:
        machine = arguments[0]
```

```
            #Вложенная функция-генератор
            def parse():
                """Возвращает объект-генератор со строкой от команды snmpdf"""
                ps = Popen([SNMPDF, "-c", options.community,
                            "-v",options.Version, machine],
                            stdout=PIPE, stderr=PIPE)
                return ps.stdout

            #Конвейер генераторов для поиска критических значений
            pattern = "9[0-9]%"
           outline = (line.split() for line in parse()) #удалить возвраты каретки
            flag = (" ".join(row) for row in outline if re.search(pattern,
                                                                  row[-1]))
            #поиск по шаблону и объединение соответствующих строк в список
            for line in flag: print "%s CRITICAL" % line
            #Пример возвращаемого значения
            #Real Memory 2067636 1974120 93516 95% CRITICAL
        else:
            p.print_help()

    if __name__ == '__main__':
        main()
```

Запустив эту «измененную» версию команды `snmpdf`, мы получили следующий результат на тестовой машине:

```
[ngift@Macintosh-8][H:10486][J:0]# python snmpdf_alter.py localhost
Real Memory 2067636 1977208 90428 95% CRITICAL
```

Теперь у нас имеется совершенно другой сценарий, который выводит только значения от 90 процентов и выше, обозначенные нами как критические. Мы могли бы запускать этот сценарий из cron каждую ночь для опроса нескольких сотен машин и отправлять результаты, полученные от нашего сценария, по электронной почте. Кроме того, мы могли бы расширить этот сценарий и отыскивать записи с объемом использования 80 процентов, 70 процентов и выдавать предупреждения по достижении этих уровней. Такой сценарий легко можно было бы объединить, например, с Google App Engine, с целью создания веб-приложения, выполняющего мониторинг использования дискового пространства во всей инфраструктуре.

Рассмотрим теперь сам программный код. Здесь есть несколько моментов, отличающих этот сценарий от предыдущих примеров, на которых стоит остановиться. Первое отличие состоит в том, что вместо функции `subprocess.call()` используется метод `subprocess.Popen()`. Если вам когда-нибудь потребуется анализировать вывод, получаемый от утилиты UNIX, то `subprocess.Popen()` – это именно то, что вам нужно. Кроме того, обратите внимание, что мы использовали метод `stdout.readlines()`, который возвращает список строк. Это будет важно позднее, когда эти выходные данные будут пропускаться через серию выражений-генераторов.

В разделе с конвейером генераторов мы пропускаем наш объект-генератор через два выражения, выполняющих поиск критических значений в соответствии с заданным нами условием. Как отмечалось выше, мы легко могли бы добавить еще пару строк с выражениями-генераторами, чтобы получить результаты для пороговых значений 70 и 80 процентов.

Этот инструмент, возможно, оказался немного более сложным, чем вам хотелось бы. Возможно, лучше было бы разбить его на несколько небольших и универсальных частей, которые можно было бы импортировать. И все-таки этот сценарий неплохо иллюстрирует наш пример.

Шаблон проектирования «гибрид кудзу»: обертывание инструментов сценариями на языке Python с порождением процессов

Наш последний пример был достаточно интересным, но существует еще один интересный способ изменения поведения существующих инструментов UNIX, основанный на запуске нескольких копий для повышения эффективности. Конечно, это может выглядеть немного странным, но иногда вам просто необходимо будет творчески подходить к своей работе. Это одна из сторон профессии системного администратора, когда время от времени для разрешения проблем приходится делать безумные вещи.

В примере этого раздела мы создали тестовый сценарий, который создавал файлы образов с помощью команд `dd`, работающих параллельно. Возьмем эту идею за основу и создадим инструмент командной строки, который можно было бы использовать снова и снова. Как минимум, получим средство создания высокой нагрузки на дисковую подсистему ввода-вывода, которое пригодится для тестирования нового файлового сервера. Исходный текст сценария приводится в примере 13.12.

Пример 13.12. Множественная команда dd

```
from subprocess import Popen, PIPE
import optparse
import sys

class ImageFile():
    """Создает файлы образов с помощью dd"""
    def __init__(self, num=None, size=None, dest=None):
        self.num = num
        self.size = size
        self.dest = dest

   def createImage(self):
        """создает N идентичных файлов образов по 10 Мбайт"""
        value = "%sMB " % str(self.size/1024)
```

```
            for i in range(self.num):
                try:
                    cmd = "dd if=/dev/zero of=%s/file.%s bs=1024 count=%s"\
                        % (self.dest,i,self.size)
                    Popen(cmd, shell=True, stdout=PIPE)
                except Exception, err:
                    sys.stderr.write(err)

        def controller(self):
            """Запускает множество команд dd"""
            p = optparse.OptionParser(description="Launches Many dd",
                                      prog="Many dd",
                                      version="0.1",
                                      usage="%prog [options] dest")
            p.add_option('-n', '--number', help='set many dd',
                         type=int)
            p.add_option('-s', '--size', help='size of image in bytes',
                         type=int)
            p.set_defaults(number=10,
                           size=10240)
            options, arguments = p.parse_args()
            if len(arguments) == 1:
                self.dest = arguments[0]
                self.size = options.size
                self.num = options.number
                #запуск команд dd
                self.createImage()

    def main():
        start = ImageFile()
        start.controller()

    if __name__ == "__main__":
        main()
```

Теперь при запуске нашей версии команды `dd` мы можем определять размер одного файла в байтах, путь и общее число файлов/процессов. Ниже показано, как выглядит вывод этого инструмента:

```
$ ./subprocess_dd.py /tmp/
$ 10240+0 records in
10240+0 records out
10485760 bytes transferred in 1.353665 secs (7746199 bytes/sec)
10240+0 records in
10240+0 records out
10485760 bytes transferred in 1.793615 secs (5846160 bytes/sec)
10240+0 records in
10240+0 records out
10485760 bytes transferred in 2.664616 secs (3935186 bytes/sec)

...дальнейший вывод опущен для экономии места....
```

Сразу же можно сказать, что этот инструмент мог бы пригодиться для тестирования производительности дисковой подсистемы ввода-вывода на высокоскоростных устройствах Fibre SAN или NAS. Приложив еще немного усилий, можно было бы добавить функции для создания отчетов в формате PDF и отправки результатов по электронной почте. Следует отметить, что то же самое можно было бы реализовать на основе потоков выполнения, если потоки соответствуют уровню сложности проблемы, которую необходимо решить.

Интеграция конфигурационных файлов

Интеграция конфигурационных файлов в инструменты командной строки имеет особую важность для повышения простоты использования и выполнения дополнительных настроек в будущем. На первый взгляд разговоры об удобстве использования инструментов командной строки кажутся немного странными, потому что обычно эта тема рассматривается только относительно приложений с графическим интерфейсом или веб-приложений. Это несправедливо, потому что инструменты командной строки заслуживают такого же внимательного отношения к простоте и удобству использования, какое уделяется при создании приложений с графическим интерфейсом.

Конфигурационный файл может также быть полезным средством для централизованного управления инструментом командной строки, запускаемым на разных машинах. Доступ к разделяемому конфигурационному файлу можно обеспечить средствами NFS, после чего сотни машин могли бы считывать его из универсального инструмента командной строки, созданного вами. С другой стороны, у вас может иметься своя система управления настройками, которая также могла бы использоваться для передачи конфигурационных файлов инструментам, созданным вами.

В состав стандартной библиотеки языка Python входит замечательный модуль `ConfigParser`, предназначенный для чтения и записи конфигурационных файлов, использующий синтаксис *ini*-файлов. Оказывается, формат *.ini* является прекрасным способом хранения простых конфигурационных данных, не требующим от человека, выполняющего редактирование файла, использования XML и знаний языка Python.

Следует иметь в виду, что порядок следования записей в конфигурационном файле не имеет значения. Для представления содержимого конфигурационного файла модуль `ConfigParser` использует словарь, и вы должны будете обращаться к нему соответствующим образом, чтобы получить корректное отображение.

Прежде чем приступить к интегрированию конфигурационных файлов в инструмент командной строки, мы создадим конфигурационный

файл «hello world». Создайте файл с именем *hello_config.ini* и добавьте в него следующие строки:

```
[Section A]
phrase=Config
```

Теперь, когда у нас имеется простейший конфигурационный файл, мы можем приступить к интегрированию этого файла в наш предыдущий пример инструмента командной строки «Hello World», как показано в примере 13.13.

Пример 13.13. Инструмент командной строки с поддержкой конфигурационного файла

```
#!/usr/bin/env python
import optparse
import ConfigParser

def readConfig(file="hello_config.ini"):
    Config = ConfigParser.ConfigParser()
    Config.read(file)
    sections = Config.sections()
    for section in sections:
        #раскомментируйте следующую строку, чтобы увидеть,
        #как выполняется разбор конфигурационного файла
        #print Config.items(section)
        phrase = Config.items(section)[0][1]
        return phrase

def main():
    p = optparse.OptionParser()
    p.add_option('--sysadmin', '-s')
    p.add_option('--config', '-c', action="store_true")
    p.set_defaults(sysadmin="BOFH")

    options, arguments = p.parse_args()
    if options.config:
        options.sysadmin = readConfig()
    print 'Hello, %s' % options.sysadmin

if __name__ == '__main__':
    main()
```

Если теперь запустить этот инструмент без каких-либо параметров, мы получим значение по умолчанию `BOFH`, как и в оригинальной версии программы «Hello World»:

```
[ngift@Macintosh-8][H:10543][J:0]# python hello_config_optparse.py
Hello, BOFH
```

Однако, если указать параметр `--config`, сценарий прочитает содержимое конфигурационного файла и даст такой ответ:

```
[ngift@Macintosh-8][H:10545][J:0]# python hello_config_optparse.py --config
Hello, Config
```

В большинстве случаев вполне достаточно будет для параметра `--config` использовать путь к файлу по умолчанию, но позволить при этом прямо указывать местоположение конфигурационного файла. Для этого вместо определения действия `store_true` можно сделать следующее:

```
p.add_option('--config', '-c',
help='Path to read in config file')
```

Если бы это была более крупная и полезная программа, мы могли бы передать ее любому пользователю, даже не знающему языка Python. Реализованный подход позволил бы ему настраивать поведение программы, изменяя значение в строке `phrase=Config`, без необходимости касаться программного кода. Даже если пользователь обладает знанием языка Python, он будет избавлен от ввода в командной строке одних и тех же параметров снова и снова, и при этом сохранится гибкость инструмента.

В заключение

Модули `optparse` и `ConfigParser`, входящие в состав стандартной библиотеки языка Python, очень просты в работе и давно уже включены в библиотеку, поэтому они должны быть доступны в большинстве систем, с которыми вам придется сталкиваться. Если вам потребуется написать множество инструментов командной строки, то есть смысл исследовать дополнительные возможности модуля `optparse`, такие как функции обратного вызова и расширение самого модуля `optparse`. Вам также могут пригодиться несколько взаимосвязанных модулей, которые отсутствуют в стандартной библиотеке, такие как: CommandLineApp (*http://www.doughellmann.com/projects/CommandLineApp/*), Argparse (*http://pypi.python.org/pypi/argparse*) и ConfigObj (*http://pypi.python.org/pypi/ConfigObj*).

14

Практические примеры

Управление DNS с помощью сценариев на языке Python

Управление сервером DNS является достаточно простой задачей по сравнению, например, с управлением конфигурационными файлами веб-сервера Apache. Программное внесение крупномасштабных изменений в DNS – вот настоящая проблема, способная сокрушать вычислительные центры и провайдеров веб-хостинга. Оказывается, в составе языка Python имеется модуль `dnspython`, который может вам пригодиться в решении подобных задач. Следует отметить, что существует еще один модуль, имеющий отношение к DNS, – `PyDNS`, но мы будем рассматривать только `dnspython`.

Обязательно ознакомьтесь с документацией, которую вы найдете на сайте *http://www.dnspython.org/*. Кроме того, существует замечательная статья об использовании модуля `dnspython`, которую вы найдете по адресу *http://vallista.idyll.org/~grig/articles/*.

Чтобы начать использовать модуль `dnspython`, вам необходимо лишь с помощью утилиты `easy_install` установить одноименный пакет из каталога пакетов Python:

```
ngift@Macintosh-8][H:10048][J:0]# sudo easy_install dnspython
Password:
Searching for dnspython
Reading http://pypi.python.org/simple/dnspython/
[дальнейший вывод обрезан]
```

Теперь попробуем исследовать модуль с помощью оболочки IPython так же, как проверялись многие другие идеи в этой книге. В следующем примере мы извлекаем записи «A» и «MX» для имени *oreilly.com*:

```
In [1]: import dns.resolver
In [2]: ip = dns.resolver.query("oreilly.com","A")
In [3]: mail = dns.resolver.query("oreilly.com","MX")
In [4]: for i,p in ip,mail:
  ....:     print i,p
  ....:
  ....:
208.201.239.37 208.201.239.36
20 smtp1.oreilly.com. 20 smtp2.oreilly.com.
```

В этом примере мы присваиваем значения записей «A» переменной `ip`, а значения записей «MX» – переменной `mail`. Результаты, полученные из записей «A», выведены в верхней строке, а результаты, полученные из записей «MX», – в нижней. Теперь, когда мы получили некоторое представление о том, как работает этот модуль, напишем сценарий, который будет получать значения записей «A» для списка хостов.

Пример 14.1. Запрос информации для группы хостов

```
import dns.resolver

hosts = ["oreilly.com", "yahoo.com", "google.com", "microsoft.com", "cnn.com"]

def query(host_list=hosts):
    collection = []
    for host in host_list:
        ip = dns.resolver.query(host,"A")
        for i in ip:
            collection.append(str(i))
    return collection

if __name__ == "__main__":
    for arec in query():
        print arec
```

Если запустить этот сценарий, будут получены значения всех записей «A» для указанных хостов, как показано ниже:

```
[ngift@Macintosh-8][H:10046][J:0]# python query_dns.py
208.201.239.37
208.201.239.36
216.109.112.135
66.94.234.13
64.233.167.99
64.233.187.99
72.14.207.99
207.46.197.32
207.46.232.182
64.236.29.120
64.236.16.20
64.236.16.52
64.236.24.12
```

Одна очевидная проблема, которую можно решить подобным образом, – программно проверить наличие корректных записей «A» для всех хостов, имена которых присутствуют в файле.

Однако модуль `dnspython` способен на большее: с его помощью можно управлять зонами DNS и выполнять более сложные запросы, чем описано здесь. Если вам интересно будет рассмотреть дополнительные примеры использования модуля, обращайтесь по адресу URL, указанному выше.

Использование протокола LDAP для работы с OpenLDAP, Active Directory и другими продуктами из сценариев на языке Python

LDAP – это новомодное словечко для большинства корпораций, а один из авторов книги даже использовал базу данных LDAP для управления своей домашней локальной сетью. Если вы не знакомы с LDAP, скажем, что эта аббревиатура расшифровывается как Lightweight Directory Access Protocol (облегченный протокол доступа к сетевому каталогу). Одно из самых удачных определений, с которыми нам приходилось сталкиваться, приводится в Википедии: «прикладной протокол, позволяющий обращаться к службе каталогов, работающий поверх протокола TCP/IP». В качестве примера одной из служб можно назвать службу аутентификации, которая, безусловно, является одним из самых популярных применений этого протокола. Примерами программных продуктов, поддерживающих протокол LDAP, могут служить Open Directory, Open LDAP, Red Hat Directory Server и Active Directory. Прикладной интерфейс python-ldap поддерживает взаимодействие с двумя продуктами – OpenLDAP и Active Directory.

Прикладной интерфейс к LDAP в языке Python называется python-ldap и включает в себя поддержку объектно-ориентированной обертки вокруг OpenLDAP 2.x. Существует также поддержка и других компонентов LDAP, включая средства обработки файлов LDIF и LDAPv3. Прежде чем начать работу с этим протоколом, вам необходимо загрузить пакет из проекта python-ldap, который находится на сайте sourceforge по адресу: http://python-ldap.sourceforge.net/download.shtml.

После установки пакета python-ldap, возможно, вам потребуется сначала ознакомиться с библиотекой в оболочке IPython. Ниже приводится протокол интерактивного сеанса, где сначала выполнена удачная попытка подключиться к общедоступному серверу LDAP, а затем неудачная попытка. Изучение особенностей установки и настройки LDAP выходит далеко за рамки этой книги и, тем не менее, мы можем начать тестировать прикладной интерфейс пакета python-ldap, используя общедоступный сервер LDAP университета штата Мичиган.

```
In [1]: import ldap

In [2]: l = ldap.open("ldap.itd.umich.edu")

In [3]: l.simple_bind()
Out[3]: 1
```

Метод `simple_bind()` сообщает нам, что соединение выполнено успешно, но давайте попробуем выполнить неудачную попытку и посмотрим, как это будет выглядеть:

```
In [5]: try:
  ....:     l = ldap.open("127.0.0.1")
  ....: except Exception,err:
  ....:     print err
  ....:
  ....:
In [6]: l.simple_bind()
---------------------------------------------------------------------------
SERVER_DOWN                               Traceback (most recent call last)

/root/<ipython console>

/usr/lib/python2.4/site-packages/ldap/ldapobject.py in simple_bind(self, who,
  cred, serverctrls, clientctrls)
    167 simple_bind([who='' [,cred='']]) -> int
    168 """
--> 169 return self._ldap_call(self._l.simple_bind,who,cred,
EncodeControlTuples(serverctrls),EncodeControlTuples(clientctrls))
    170
    171 def simple_bind_s(self,who='',cred='',serverctrls=None,
clientctrls=None):

/usr/lib/python2.4/site-packages/ldap/ldapobject.py in _ldap_call(self, func,
  *args, **kwargs)
     92     try:
     93         try:
---> 94             result = func(*args,**kwargs)
     95         finally:
     96             self._ldap_object_lock.release()

SERVER_DOWN: {'desc': "Can't contact LDAP server"}
```

Как видно из этого примера, наш программный код не нашел запущенный сервер LDAP и разразился ругательствами.

Импортирование файла LDIF

Простое подключение к общедоступному серверу LDAP не настолько полезная операция, чтобы помочь нам в нашей работе. Ниже приводится пример выполнения асинхронного импорта LDIF:

```
import ldap
import ldap.modlist as modlist
```

```
ldif = "somefile.ldif"
def create():
    l = ldap.initialize("ldaps://localhost:636/")
    l.simple_bind_s("cn=manager,dc=example,dc=com","secret")
    dn="cn=root,dc=example,dc=com"
    rec = {}
    rec['objectclass'] = ['top','organizationalRole','simpleSecurityObject']
    rec['cn'] = 'root'
    rec['userPassword'] = 'SecretHash'
    rec['description'] = 'User object for replication using slurpd'
    ldif = modlist.addModlist(attrs)
    l.add_s(dn,ldif)
    l.unbind_s()
```

В этом примере мы сначала инициализируем соединение с локальным сервером LDAP, затем создаем объект, который будет служить проекцией базы данных LDAP, и потребуется, когда мы будем выполнять асинхронный импорт файла LDIF. Обратите внимание, что использование метода `l.add_s()` указывает, что выполняется асинхронное обращение к прикладному интерфейсу.

Это лишь самые основы совместного использования LDAP и Python, а за дополнительной информацией об использовании пакета python-ldap вам следует обращаться к ресурсу, указанному в начале этого раздела. Там, в частности, вы найдете примеры использования LDAPv3 – Create, Read, Update, Delete (CRUD – создание, чтение, изменение и удаление) и многие другие.

И последнее, о чем хотелось бы упомянуть: для языка Python существует инструмент с названием web2ldap, который реализует веб-интерфейс к LDAP и разработан автором пакета python-ldap. Возможно, у вас появится желание опробовать его наряду с другими альтернативными решениями управления LDAP через веб-интерфейс. Перейдя по адресу *http://www.web2ldap.de/*, вы найдете официальную документацию к этому инструменту, которая очень подробно описывает поддержку LDAPv3.

Составление отчета на основе файлов журналов Apache

В настоящее время доля веб-сервера Apache составляет примерно 50 процентов от всех веб-серверов в Интернете. Цель следующего примера состоит в том, чтобы показать вам способ составления отчетов на основе файлов журналов веб-сервера Apache. В этом примере рассматривается только часть информации, доступной в файлах журналов Apache, но вы можете использовать описываемый подход для извлечения любых данных, содержащихся в этих файлах журналов. Данный подход можно легко адаптировать для работы с огромными файлами данных и для работы с большим числом данных.

В главе 3 приводилось несколько примеров анализа файлов журналов веб-сервера Apache, из которых извлекалась некоторая информация. В этом примере мы повторно воспользуемся модулями, написанными для главы 3, чтобы продемонстрировать, как создавать удобочитаемые отчеты из одного или более файлов журналов. Помимо обработки всех файлов журналов, список которых определяется отдельно, вы можете указать этому сценарию, что он должен объединить файлы журналов и создать единый отчет. Исходный текст сценарий приводится в примере 14.2.

Пример 14.2. Объединенный отчет на основе файлов журналов веб-сервера Apache

```
#!/usr/bin/env python

from optparse import OptionParser

def open_files(files):
    for f in files:
        yield (f, open(f))

def combine_lines(files):
    for f, f_obj in files:
        for line in f_obj:
            yield line

def obfuscate_ipaddr(addr):
    return ".".join(str((int(n) / 10) * 10) for n in addr.split('.'))

if __name__ == '__main__':
    parser = OptionParser()
    parser.add_option("-c", "--consolidate", dest="consolidate",
                      default=False,
                      action='store_true', help="consolidate log files")
    parser.add_option("-r", "--regex", dest="regex", default=False,
                      action='store_true', help="use regex parser")

    (options, args) = parser.parse_args()
    logfiles = args

    if options.regex:
        from apache_log_parser_regex import generate_log_report
    else:
        from apache_log_parser_split import generate_log_report

    opened_files = open_files(logfiles)

    if options.consolidate:
        opened_files = (('CONSOLIDATED', combine_lines(opened_files)),)

    for filename, file_obj in opened_files:
        print "*" * 60
        print filename
        print "-" * 60
        print "%-20s%s" % ("IP ADDRESS", "BYTES TRANSFERRED")
```

```
print "-" * 60
report_dict = generate_log_report(file_obj)
for ip_addr, bytes in report_dict.items():
    print "%-20s%s" % (obfuscate_ipaddr(ip_addr), sum(bytes))
print "=" * 60
```

В самом начале сценария определяются две функции: `open_files()` и `combine_lines()`. Позднее обе эти функции будут использоваться в генераторах для упрощения программного кода. Функция `open_files()` – это функция-генератор, которая принимает список (в действительности – любой итерируемый объект) имен файлов. Для каждого имени файла она создает кортеж из имени файла и соответствующего ему объекта открытого файла. Функция `combine_lines()` принимает итерируемые объекты открытых файлов в виде единственного аргумента. Она выполняет обход объектов файлов в цикле `for`. Для каждого файла выполняется обход строк в этом файле. И на каждой итерации она – с помощью инструкции `yield` – возвращает очередную строку. Итерируемый объект, получаемый от функции `combine_lines()`, можно сравнить с файлом: мы можем выполнять обход строк в этом объекте.

Затем с помощью модуля `optparse` выполняется разбор аргументов командной строки, полученных от пользователя. Мы будем принимать только два аргумента, оба – логического типа: признак объединения файлов журналов и признак необходимости использовать библиотеку регулярных выражений. Параметр `consolidate` сообщает сценарию, что все файлы должны быть объединены при составлении отчета. Если сценарию передается этот параметр, мы, в некотором смысле, выполняем конкатенацию содержимого файлов. Но к этому мы еще вернемся. Параметр `regex` сообщает сценарию, что вместо библиотеки «split» следует использовать библиотеку регулярных выражений, которая была написана нами в главе 3. Обе они предлагают идентичные функциональные возможности, но библиотека «split» работает быстрее.

Затем проверяется, был ли указан параметр `regex`. Если параметр был указан, то импортируется модуль `apache_log_parser_regex`. В противном случае используется модуль `apache_log_parser_split`. В действительности мы включили этот параметр, чтобы сравнить производительность двух библиотек. О производительности этого сценария мы поговорим немного позже.

Затем вызывается функция `open_files()`, которой передается список имен файлов, полученный от пользователя. Как мы уже упоминали, функция `open_files()` – это функция-генератор, которая возвращает объект файла для каждого имени во входном списке. Это означает, что каждый файл открывается фактически, только когда функция возвращает соответствующий объект. Теперь, когда у нас имеется итерируемый объект с открытыми файлами, мы можем выполнять с ним некоторые операции. Мы можем выполнить обход всех файлов и составить отчет по каждому из них или объединить их некоторым способом и со-

ставить объединенный отчет сразу по всем файлам. Это как раз то место, где на сцену выходит функция `combine_lines()`. Если пользователь передал ключ «consolidate», то «список файлов», по которому будут выполняться итерации, будет содержать единственный объект, подобный файлу: генератор всех строк во всех файлах.

Далее, независимо от того, настоящие файлы содержатся в списке или комбинированный файл, каждый из них передается соответствующей функции `generate_log_report()`, которая возвращает словарь с IP-адресами и количеством байтов, отправленных по этим адресам. Для каждого файла выводятся строки-разделители и отформатированные строки с результатами работы функции `generate_log_report()`. Ниже приводится вывод, полученный в результате обработки одного файла журнала размером 28 Кбайт:

```
**********************************************************
access.log
----------------------------------------------------------
IP ADDRESS            BYTES TRANSFERRED
----------------------------------------------------------
190.40.10.0           17479
200.80.230.0          45346
200.40.90.110         8276
130.150.250.0         0
70.0.10.140           2115
70.180.0.220          76992
200.40.90.110         23860
190.20.250.190        499
190.20.250.210        431
60.210.40.20          27681
60.240.70.180         20976
70.0.20.120           1265
190.20.250.210        4268
190.50.200.210        4268
60.100.200.230        0
70.0.20.190           378
190.20.250.250        5936
==========================================================
```

Вывод, полученный в результате обработки трех файлов журналов (фактически это три копии одного и того же файла, созданные путем многократного копирования данных из оригинального файла), выглядит, как показано ниже:

```
**********************************************************
access.log
----------------------------------------------------------
IP ADDRESS            BYTES TRANSFERRED
----------------------------------------------------------
190.40.10.0           17479
200.80.230.0          45346
```

```
<обрезано>
70.0.20.190           378
190.20.250.250        5936
============================================================
************************************************************
access_big.log
------------------------------------------------------------
IP ADDRESS            BYTES TRANSFERRED
------------------------------------------------------------
190.40.10.0           1747900
200.80.230.0          4534600
<обрезано>
70.0.20.190           37800
190.20.250.250        593600
============================================================
************************************************************
access_bigger.log
------------------------------------------------------------
IP ADDRESS            BYTES TRANSFERRED
------------------------------------------------------------
190.40.10.0           699160000
200.80.230.0          1813840000
<обрезано>
70.0.20.190           15120000
190.20.250.250        237440000
============================================================
```

А ниже приводится объединенный отчет для всех трех файлов:

```
************************************************************
CONSOLIDATED
------------------------------------------------------------
IP ADDRESS            BYTES TRANSFERRED
------------------------------------------------------------
190.40.10.0           700925379
200.80.230.0          1818419946
<обрезано>
190.20.250.250        238039536
============================================================
```

Итак, какова же производительность этого сценария? И каково потребление памяти? Все тесты, которые приводятся в этом разделе, выполнялись на сервере Ubuntu Gutsy, с процессором Athlon 64 X2 5400+ 2.8 ГГц, с объемом ОЗУ 2 Гбайта и с жестким диском Seagate Barracuda 7200 RPM SATA. Размер файла журнала составлял примерно 1 Гбайт:

```
jmjones@ezr:/data/logs$ ls -l access*log
-rw-r--r-- 1 jmjones jmjones 1157080000 2008-04-18 12:46 access_bigger.log
```

Ниже приводятся результаты тестирования:

```
$ time python summarize_logfiles.py --regex access_bigger.log
```

```
************************************************************
access_bigger.log
------------------------------------------------------------
IP ADDRESS            BYTES TRANSFERRED
------------------------------------------------------------
190.40.10.0           699160000
<обрезано>
190.20.250.250        237440000
============================================================
real 0m46.296s
user 0m45.547s
sys 0m0.744s

jmjones@ezr:/data/logs$ time python summarize_logfiles.py access_bigger.log
************************************************************
access_bigger.log
------------------------------------------------------------
IP ADDRESS            BYTES TRANSFERRED
------------------------------------------------------------
190.40.10.0           699160000
<обрезано>
190.20.250.250        237440000
============================================================
real 0m34.261s
user 0m33.354s
sys 0m0.896s
```

При использовании библиотеки, выполняющей извлечение данных с помощью регулярных выражений, на создание отчета ушло порядка 46 секунд. При использовании версии, использующей метод `string.split()`, на создание отчета ушло 34 секунды. Но показатели потребления памяти оказались плачевными. Объем занятой памяти достиг 130 Мбайт. Причина в том, что функция `generate_log_report()` сохраняет список переданных байтов для каждого IP-адреса в файле журнала. Поэтому, чем больше файл, тем больший объем памяти будет потреблять этот сценарий. Но мы можем с этим кое-что сделать. Ниже приводится менее «жадная до памяти» версия библиотеки, выполняющей анализ файла журнала:

```
#!/usr/bin/env python

def dictify_logline(line):
    '''возвращает словарь, содержащий информацию, извлеченную
    из комбинированного файла журнала

    В настоящее время нас интересуют только адреса удаленных хостов
    и количество переданных байтов, но в качестве дополнительной меры
    мы добавили выборку кода состояния.
    '''
    split_line = line.split()
    return {'remote_host': split_line[0],
            'status': split_line[8],
```

```
            'bytes_sent': split_line[9],
        }

    def generate_log_report(logfile):
        '''возвращает словарь в формате:
           remote_host=>[список числа переданных байтов]

        Эта функция принимает объект типа file, выполняет обход всех строк
        в файле и создает отчет о количестве байтов, переданных
        при каждом обращении удаленного хоста к веб-серверу.
        '''
        report_dict = {}
        for line in logfile:
            line_dict = dictify_logline(line)
            host = line_dict['remote_host']
            #print line_dict
            try:
                bytes_sent = int(line_dict['bytes_sent'])
            except ValueError:
                ##полностью игнорировать непонятные нам ошибки
                continue
            report_dict[host] = report_dict.setdefault(host, 0) + bytes_sent
        return report_dict
```

Теперь подсчет общего числа переданных байтов ведется по мере извлечения значений, а не в вызывающей функции. Ниже приводится несколько измененная версия сценария `summarize_logfiles` **с новым параметром, позволяющим импортировать библиотеку с пониженным потреблением памяти:**

```
#!/usr/bin/env python

from optparse import OptionParser

def open_files(files):
    for f in files:
        yield (f, open(f))

def combine_lines(files):
    for f, f_obj in files:
        for line in f_obj:
            yield line

def obfuscate_ipaddr(addr):
    return ".".join(str((int(n) / 10) * 10) for n in addr.split('.'))

if __name__ == '__main__':
    parser = OptionParser()
    parser.add_option("-c", "--consolidate", dest="consolidate",
                      default=False,
                      action='store_true', help="consolidate log files")
    parser.add_option("-r", "--regex", dest="regex", default=False,
                      action='store_true', help="use regex parser")
    parser.add_option("-m", "--mem", dest="mem", default=False,
```

```
                    action='store_true', help="use mem parser")

(options, args) = parser.parse_args()
logfiles = args

if options.regex:
    from apache_log_parser_regex import generate_log_report
elif options.mem:
    from apache_log_parser_split_mem import generate_log_report
else:
    from apache_log_parser_split import generate_log_report

opened_files = open_files(logfiles)

if options.consolidate:
    opened_files = (('CONSOLIDATED', combine_lines(opened_files)),)

for filename, file_obj in opened_files:
    print "*" * 60
    print filename
    print "-" * 60
    print "%-20s%s" % ("IP ADDRESS", "BYTES TRANSFERRED")
    print "-" * 60
    report_dict = generate_log_report(file_obj)
    for ip_addr, bytes in report_dict.items():
        if options.mem:
            print "%-20s%s" % (obfuscate_ipaddr(ip_addr), bytes)
        else:
            print "%-20s%s" % (obfuscate_ipaddr(ip_addr), sum(bytes))
    print "=" * 60
```

Эти изменения привели к тому, что сценарий стал выполняться немного быстрее, чем версия с большим потреблением памяти:

```
jmjones@ezr:/data/logs$ time ./summarize_logfiles_mem.py --mem
access_bigger.log
************************************************************
access_bigger.log
------------------------------------------------------------
IP ADDRESS            BYTES TRANSFERRED
------------------------------------------------------------
190.40.10.0           699160000
<snip>
190.20.250.250        237440000
============================================================
real 0m30.508s
user 0m29.866s
sys 0m0.636s
```

На протяжении работы этого сценария потребление памяти составило порядка 4 Мбайт. Этот сценарий способен обрабатывать 2 Гбайтные файлы журналов за одну минуту. Теоретически размеры файлов могут быть неопределенно большого размера, и это не будет приводить к существенному увеличению объемов потребляемой памяти, как в преды-

дущей версии. Однако, поскольку для хранения данных используется словарь, каждый ключ которого – это уникальный IP-адрес, потребление памяти будет расти с увеличением числа уникальных IP-адресов. Если объем потребляемой памяти станет слишком велик, вы могли бы заменить словарь каким-нибудь хранилищем данных, или даже реляционной базой данных, такой как Berkeley DB.

Зеркало FTP

Следующий пример показывает, как соединяться с сервером FTP и рекурсивно получать все файлы с этого сервера, начиная с некоторого каталога, определяемого пользователем. Кроме того, этот сценарий позволяет удалять файлы после того, как они были получены. Вы можете задаться вопросом: «Зачем нужен такой сценарий? Разве все это нельзя сделать с помощью rsync?». Ответ на него: «Да, это так». Однако как быть, если утилита `rsync` отсутствует на сервере, где вы работаете, и у вас недостаточно прав, чтобы установить ее? (Это необычно для системного администратора, но такое тоже бывает.) Или как быть, если у вас нет доступа к серверу, откуда вы пытаетесь получить файлы, через SSH или rsync? В таких ситуациях данный сценарий будет служить альтернативой. Исходный текст сценария зеркалирования приводится ниже:

```
#!/usr/bin/env python

import ftplib
import os

class FTPSync(object):
    def __init__(self, host, username, password, ftp_base_dir,
                 local_base_dir, delete=False):

        self.host = host
        self.username = username
        self.password = password
        self.ftp_base_dir = ftp_base_dir
        self.local_base_dir = local_base_dir
        self.delete = delete

        self.conn = ftplib.FTP(host, username, password)
        self.conn.cwd(ftp_base_dir)
        try:
            os.makedirs(local_base_dir)
        except OSError:
            pass
        os.chdir(local_base_dir)

    def get_dirs_files(self):
        dir_res = []
        self.conn.dir('.', dir_res.append)
        files = [f.split(None, 8)[-1] for f in dir_res if f.startswith('-')]
```

```
            dirs = [f.split(None, 8)[-1] for f in dir_res if f.startswith('d')]
            return (files, dirs)

        def walk(self, next_dir):
            print "Walking to", next_dir
            self.conn.cwd(next_dir)
            try:
                os.mkdir(next_dir)
            except OSError:
                pass
            os.chdir(next_dir)

            ftp_curr_dir = self.conn.pwd()
            local_curr_dir = os.getcwd()

            files, dirs = self.get_dirs_files()
            print "FILES:", files
            print "DIRS:", dirs
            for f in files:
                print next_dir, ':', f
                outf = open(f, 'wb')
                try:
                    self.conn.retrbinary('RETR %s' % f, outf.write)
                finally:
                    outf.close()
                if self.delete:
                    print "Deleting", f
                    self.conn.delete(f)
            for d in dirs:
                os.chdir(local_curr_dir)
                self.conn.cwd(ftp_curr_dir)
                self.walk(d)

        def run(self):
            self.walk('.')

    if __name__ == '__main__':
        from optparse import OptionParser
        parser = OptionParser()
        parser.add_option("-o", "--host", dest="host",
                          action='store', help="FTP host")
        parser.add_option("-u", "--username", dest="username",
                          action='store', help="FTP username")
        parser.add_option("-p", "--password", dest="password",
                          action='store', help="FTP password")
        parser.add_option("-r", "--remote_dir", dest="remote_dir",
                          action='store', help="FTP remote starting directory")
        parser.add_option("-l", "--local_dir", dest="local_dir",
                          action='store', help="Local starting directory")
        parser.add_option("-d", "--delete", dest="delete", default=False,
                          action='store_true', help="use regex parser")

        (options, args) = parser.parse_args()
        f = FTPSync(options.host, options.username, options.password,
```

```
                options.remote_dir, options.local_dir, options.delete)
    f.run()
```

Этот сценарий выглядит проще с использованием класса. Конструктор класса принимает несколько параметров. Чтобы соединиться с сервером, конструктору необходимо передать имя удаленного хоста (`host`), имя пользователя (`username`) и пароль (`password`). Параметры `ftp_base_dir` и `local_base_dir` передаются, чтобы переместиться в требуемые каталоги на стороне сервера и на стороне локального компьютера. Параметр `delete` – это обычный флаг, который указывает, требуется ли удалять файлы на удаленном сервере после их загрузки. В определении конструктора видно, что этому параметру по умолчанию присваивается значение `False`.

После установки этих значений в виде атрибутов объекта выполняется соединение с указанным сервером FTP и производится регистрация. Затем осуществляется переход в начальный каталог на стороне сервера и в начальный каталог на локальном компьютере. Прежде чем выполнить переход в требуемый каталог на локальном компьютере, сценарий сначала пытается создать его. Если каталог уже существует, будет получено исключение `OSError`, которое игнорируется сценарием.

В классе определяются три дополнительных метода: `get_dirs_files()`, `walk()` и `run()`. Метод `get_dirs_files()` определяет, какие файлы находятся в текущем каталоге и какие из них являются обычными файлами, а какие каталогами. (К слову сказать, такой способ будет работать только в случае сервера, работающего под управлением UNIX.) Определение, какие из файлов являются обычными файлами, а какие каталогами, производится по первому символу в строках полученного списка. Если первый символ `d`, значит, – это каталог. Если первый символ '-', значит, – это файл. Благодаря этому сценарий не будет следовать за символическими ссылками или заниматься обработкой блочных устройств.

Следующий метод, который определен в классе, – это метод `walk()`. В этом методе выполняется вся основная работа. Метод `walk()` принимает единственный параметр: следующий каталог, который требуется посетить. Прежде чем двинуться дальше, напомним, что это рекурсивная функция. Она будет вызывать саму себя. Если какой-либо каталог содержит другие каталоги, метод `walk()` также обойдет их. Метод `walk()` сначала переходит в указанный каталог на стороне сервера. Затем выполняется переход в одноименный каталог на локальном компьютере, при этом, в случае необходимости, каталог создается. Потом текущая позиция на сервере FTP и на локальном компьютере сохраняется в переменных `ftp_curr_dir` и `local_curr_dir` для последующего использования. Далее с помощью метода `get_dirs_files()`, о котором уже говорилось выше, производится получение списков файлов и каталогов. Загрузка каждого файла в каталоге производится с помощью метода FTP `retrbinary()`. Кроме того, если был установлен флаг `delete`,

выполняется удаление файла. Затем выполняется переход в текущие каталоги на стороне сервера FTP и на локальном компьютере и вызывается метод `walk()` для обхода нижележащих каталогов. Переход в текущие каталоги выполняется для того, чтобы при возвращении из рекурсивных вызовов метода `walk()` мы оказались в том же самом месте, где и были.

Последний метод, который определен в классе, – это метод `run()`. Метод `run()` создан исключительно для удобства. Он просто вызывает метод `walk()` и передает ему текущий каталог FTP.

В сценарии предусмотрена только самая необходимая обработка ошибок и исключительных ситуаций. Во-первых, сценарий не проверяет правильность аргументов командной строки и поэтому пользователь должен обеспечить передачу, по крайней мере, трех параметров – имени удаленного хоста, имени пользователя и пароля. Если какой-либо из этих параметров будет отсутствовать, сценарий очень быстро завершится с сообщением об ошибке. Кроме того, если произошло исключение, сценарий не повторяет попытку загрузить файл. То есть, если что-то будет препятствовать загрузке файла, мы получим исключение, и работа программы на этом завершится. Если сценарий завершит работу на полпути, во время загрузки файлов, то при следующем запуске сценарий повторно загрузит файлы, которые уже были загружены. В такой реализации есть свой плюс, который состоит в том, что если файл был загружен только частично, он не будет удален на стороне сервера.

Приложение
Функции обратного вызова

Концепция функций обратного вызова и передачи функций в виде параметров может оказаться вам незнакомой. Если это так, то вам определенно стоит углубиться в ее изучение, чтобы понять ее достаточно хорошо для применения на практике или, по крайней мере, настолько, чтобы понимать, что происходит в сценарии, когда вы будете встречать ее. В языке Python функции являются «обычными» объектами, то есть вы можете передавать их и обращаться с ними как с объектами, потому что они действительно являются объектами. Рассмотрим пример 1.

Пример 1. Функции – типичные объекты

```
In [1]: def foo():
   ...:     print foo
   ...:
   ...:

In [2]: foo
Out[2]: <function foo at 0x1233270>

In [3]: type(foo)
Out[3]: <type 'function'>

In [4]: dir(foo)
Out[4]:
['__call__',
'__class__',
'__delattr__',
'__dict__',
'__doc__',
'__get__',
'__getattribute__',
'__hash__',
'__init__',
'__module__',
'__name__',
'__new__',
'__reduce__',
'__reduce_ex__',
```

```
'__repr__',
'__setattr__',
'__str__',
'func_closure',
'func_code',
'func_defaults',
'func_dict',
'func_doc',
'func_globals',
'func_name']
```

Простое обращение к функции, такой как `foo` из предыдущего примера, не приводит к ее вызову. Ссылаясь на имя функции, можно получать значения любых атрибутов функции, которые она имеет, и даже обращаться к функции по другому имени, как показано в примере 2.

Пример 2. Обращение к функции по имени

```
In [1]: def foo():
   ...:     """это строка документирования"""
   ...:     print "IN FUNCTION FOO"
   ...:
   ...:

In [2]: foo
Out[2]: <function foo at 0x8319534>

In [3]: foo.__doc__
Out[3]: 'this is a docstring'

In [4]: bar = foo

In [5]: bar
Out[5]: <function foo at 0x8319534>

In [6]: bar.__doc__
Out[6]: 'this is a docstring'

In [7]: foo.a = 1

In [8]: bar.a
Out[8]: 1

In [9]: foo()
IN FUNCTION FOO

In [10]: bar()
IN FUNCTION FOO
```

Здесь была создана новая функция `foo`, так чтобы она содержала строку документирования. После этого мы заявили, что переменная `bar` будет указывать на только что созданную функцию `foo`. В языке Python то, что вы привыкли считать переменными, в действительности является просто именами, указывающими (или ссылающимися) на некоторые объекты. Процесс присваивания имени объекту называется «связыванием имени». Поэтому, когда мы создали функцию `foo`, на самом

деле мы создали объект функции, а затем связали его с именем `foo`. Воспользовавшись интерактивной оболочкой IPython, чтобы получить основную информацию об имени `foo`, мы получили сообщение о том, что это функция `foo`. Интересно то, что оболочка сказала то же самое и об имени `bar`, а именно, что это функция `foo`. Мы установили значение атрибута функции `foo` и сумели обратиться к нему с помощью имени `bar`. А вызов по именам `foo` и `bar` дал одинаковые результаты.

Одно из мест в этой книге, где мы используем функции обратного вызова, – это глава 5 «Сети». Передача функций в качестве параметров, как это сделано в указанной главе в примере, демонстрирующем использование модуля `ftplib`, обеспечивает высокий динамизм во время выполнения и гибкость во время разработки и может даже расширять возможности повторной используемости программного кода. Даже если вы полагаете, что вам никогда не придется использовать функции обратного вызова, сама перестройка процесса мышления при добавлении этих знаний к вашему мыслительному арсеналу представляет большую ценность.

Алфавитный указатель

E

F

G

H

I

J

L

M

N

O

P

R

S

T

Д

З

И

К

M

H

O

Т

У

Ф

Ч

Ш

Э

Я