O'REILLY®

Авто-стопом по Python

THE HITCHHIKER'S GUIDE TO PYTHON

Кеннет Рейтц
Таня Шлюссер

ПИТЕР®

The Hitchhiker's Guide to Python

Best Practices for Development

Kenneth Reitz and Tanya Schlusser

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

Кеннет Рейтц, Таня Шлюссер

Автостопом по Python

ПИТЕР®
Санкт-Петербург • Москва • Екатеринбург • Воронеж
Нижний Новгород • Ростов-на-Дону • Самара • Минск
2017

Кеннет Рейтц, Таня Шлюссер

Автостопом по Python

Серия «Бестселлеры O'Reilly»

Перевел с английского Е. Зазноба

Заведующая редакцией	*Ю. Сергиенко*
Руководитель проекта	*О. Сивченко*
Ведущий редактор	*Н. Гринчик*
Литературный редактор	*О. Андросик*
Художественный редактор	*С. Заматевская*
Корректоры	*О. Андриевич, Т. Курьянович*
Верстка	*А. Барцевич*

ББК 32.973.2-018.1
УДК 004.43

Рейтц К., Шлюссер Т.

Р35 Автостопом по Python. — СПб.: Питер, 2017. — 336 с.: ил. — (Серия «Бестселлеры O'Reilly»).

ISBN 978-5-496-03023-6

Перед вами — увлекательная книга, которую по достоинству оценит любой неравнодушный программист и даже бывалый питонщик. Она составлена на основе одноименного онлайнового руководства http://docs.python-guide.org/en/latest/ и содержит наработки многочисленных профессионалов и энтузиастов, знающих, что такое Python и чего вы от него хотите. Проверенные методы и новейшие приемы, собранные в этой книге, помогут вам стать профессиональным Python-программистом и во всеоружии встретить наступающую эпоху Python 3.

12+ (В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

ISBN 978-1491933176 англ.

ISBN 978-5-496-03023-6

Изготовлено в России. Изготовитель: ООО «Питер Пресс». Место нахождения и фактический адрес: 192102, Россия, город Санкт-Петербург, улица Андреевская, дом 3, литер А, помещение 7Н. Тел.: +78127037373.

Дата изготовления: 06.2017. Наименование: книжная продукция. Срок годности: не ограничен.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12 — Книги печатные профессиональные, технические и научные.

Подписано в печать 31.05.17. Формат 70×100/16. Бумага офсетная. Усл. п. л. 27,090. Тираж 1200. Заказ 0000.

Краткое содержание

Приложение. Дополнительная информация

Оглавление

Часть II. Переходим к делу

Часть III. Руководство по сценариям

Приложение. Дополнительная информация

Введение

Python большой. Действительно большой. Вы не поверите своим глазам, когда увидите, насколько он огромен.

Это руководство *не* предназначено для того, чтобы обучить вас языку Python (мы приведем ссылки на множество хороших ресурсов, которые помогут вам в этом), оно скорее представляет собой (безапелляционное) руководство от специалиста, где рассматриваются популярные инструменты и лучшие практики нашего сообщества. Аудитория этой книги разнообразна — от новичков до программистов Python среднего уровня, которые либо хотели бы внести свой вклад в развитие программного обеспечения (ПО) с открытым исходным кодом, либо начинают карьеру или создают компанию и собираются писать на Python (однако для рядовых пользователей Python также будут полезными часть I и глава 5).

В первой части книги мы поговорим о том, как выбрать текстовый редактор или интерактивную среду разработки, которые подойдут вам для работы (например, читатели, которые часто используют язык Java, могут предпочесть Eclipse с встроенным плагином для Python). Кроме того, рассматриваются другие интерпретаторы, удовлетворяющие те потребности, в отношении которых вы даже предположить не могли, что Python может с этим справиться (например, существует реализация MycroPython, основанная на чипе ARM Cortex-M4). Во второй части демонстрируется «питонский» стиль выделения кода примеров, принятый в сообществе, работающем с открытым исходным кодом. Надеемся, этот стиль вдохновит вас на углубленное изучение и экспериментирование с открытым кодом. В третьей части кратко рассматривается широкий перечень библиотек, наиболее часто используемых в сообществе Python, — это поможет вам получить представление о том, какие задачи Python может решать в текущий момент.

Условные обозначения

В этой книге используются следующие условные обозначения.

Курсив

Курсивом выделены новые термины.

`Моноширинный шрифт`

Применяется для листингов программ, а также внутри абзацев, чтобы обратиться к элементам программы вроде переменных, функций и типов данных. Им также выделены имена и расширения файлов.

`Полужирный моноширинный шрифт`

Показывает команды или другой текст, который пользователь должен ввести самостоятельно.

`Курсивный моноширинный шрифт`

Показывает текст, который должен быть заменен значениями, введенными пользователем, или значениями, определяемыми контекстом.

Шрифт без засечек

Используется для обозначения URL-адресов, адресов электронной почты.

Этот рисунок указывает на совет или предложение.

Этот рисунок указывает на общее замечание.

Этот рисунок указывает на предупреждение.

Благодарности

Друзья, добро пожаловать на страницы книги «Автостопом по Python»!

Эта книга, насколько мне известно, первая в своем роде: она создана одним автором (мной — Kenneth), а большая часть содержимого предоставлена сотнями людей со всего света бесплатно. В истории человечества никогда не было такой технологии, которая позволила бы осуществиться совместному проекту такого размера и масштаба.

Создание этой книги стало возможным благодаря:

- *сообществу* — любовь объединяет нас, позволяя преодолеть все препятствия;
- *программным проектам* Python, Sphinx, Alabaster и Git;
- *сервисам* GitHub и Read the Docs.

Наконец, я хотел бы лично поблагодарить Таню (Tanya Schlusser), проделавшую сложную работу по преобразованию моего труда в форму книги и подготовившую ее к публикации, а также потрясающую команду издательства O'Reilly — Доуна (Dawn), Жасмин (Jasmine), Ника (Nick), Хезер (Heather), Николь (Nicole), Мег (Meg) и десятки людей, которые работали для того, чтобы эта книга стала еще лучше.

Часть I. Приступаем

Эта часть посвящена настройке среды Python. Нас вдохновило руководство по Python для Windows автора Стюарт Элисс (Stuart Eliss) (http://www.stuartellis.eu/articles/python-development-windows/). Первая часть состоит из следующих глав.

- Глава 1 «Выбираем интерпретатор». Мы сравним Python 2 и Python 3, а также рассмотрим некоторые интерпретаторы в качестве альтернативы CPython.
- Глава 2 «Правильная установка Python». Мы покажем, как получить Python, а также инструменты pip и virtualenv.
- Глава 3 «Ваша среда разработки». Мы опишем наши любимые текстовые редакторы и IDE для разработки с помощью Python.

1 Выбираем интерпретатор

Python 2 против Python 3

При выборе интерпретатора для Python всегда возникает вопрос, что выбрать — Python 2 или Python 3? Ответ не всегда очевиден, однако третья версия с каждым днем становится все более популярной.

Дела обстоят следующим образом:

- долгое время стандартом был Python 2.7;
- в Python 3 произошли значительные изменения в языке (этим немного недовольны некоторые разработчики[1]);
- Python 2.7 будет получать обновления безопасности до 2020 года (https://www.python.org/dev/peps/pep-0373/);
- Python 3 продолжает развиваться, как это происходило с Python 2 в последние годы.

Теперь вы понимаете, почему этот выбор не из простых.

---

[1] Если вы не работаете с сетями на низком уровне, то практически не заметите изменений. Увидите только то, что выражение print стало функцией. В противном случае фраза «немного недовольны» является преуменьшением: разработчики, ответственные за крупные и популярные библиотеки для работы с Интернетом, сокетами или сетями, которые взаимодействуют со строками, представленными в формате Unicode и байтовом формате, должны были (и все еще должны) внести значительные изменения. Подробнее об изменениях, начавшихся, когда Python 3 явился миру, можно узнать из статьи под названием Everything you thought you knew about binary data and Unicode has changed (http://bit.ly/text-vs-data).

Рекомендации

Как нам кажется, опытный разработчик скорее выберет Python 3. Но, если вы можете работать только с Python 2, вы все равно будете считаться пользователем Python. Вот наши рекомендации.

Используйте Python 3, если...

- вам нравится Python 3;
- вы не знаете, какую версию выбрать;
- вы новатор.

Используйте Python 2, если...

- вам нравится Python 2 и вас печалит мысль, что в будущем вас ждет лишь Python 3;
- это повлияет на требования к стабильности вашего приложения[1];
- этого требует программное обеспечение, от которого вы зависите.

То есть... Python 3?

Если вы выбираете интерпретатор для Python и не имеете предубеждений, вам следует использовать новейший Python 3.x (в каждой версии появляются новые и улучшенные модули стандартных библиотек, а также исправления ошибок и пробелов в безопасности). Прогресс не стоит на месте. Выбирайте Python 2, только если на то есть серьезная причина, например вы работаете с библиотекой, которая использует Python 2 и не имеет адекватной альтернативы в Python 3, если вам нужна конкретная реализация (см. раздел «Реализации» далее) или если вам просто нравится работать с Python 2 (как некоторым из нас).

С помощью ресурса Can I Use Python 3? (https://caniusepython3.com/) вы можете проверить, блокируют ли используемые вами проекты возможность работать с Python 3.

Для получения информации обратитесь к Python2orPython3 (http://bit.ly/python2-or-python3), где указываются причины обратной несовместимости в спецификации языков, а также приводятся ссылки на подробные описания различий.

Если вы только начинаете работу с Python, кое о чем вам следует волноваться больше, чем о кросс-совместимости всех версий Python. Просто пишите работающий код для своей системы, а остальное наверстаете позже.

---

[1] Высокоуровневый список изменений вы найдете по адресу http://python3porting.com/stdlib.html в стандартной библиотеке Python (Python's Standard Library).

Реализации

Когда говорят о Python, зачастую имеют в виду не только сам язык, но и реализацию CPython. По сути, Python — это спецификация для языка, которая может быть реализована множеством разных способов.

Разные реализации могут пригодиться для совместимости с другими библиотеками или же для небольшого ускорения. Библиотеки Python должны работать независимо от вашей реализации Python, но работоспособность библиотек, основанных на языке C (например, NumPy), не гарантируется. В этом разделе мы рассмотрим наиболее популярные реализации Python.

В этом руководстве подразумевается, что вы работаете со стандартной реализацией CPython для Python 3, однако мы часто будем делать пометки для Python 2.

CPython

CPython (http://www.python.org/) — это базовая реализация[1] Python, написанная на языке C. Она компилирует код Python в промежуточный байт-код, который затем интерпретируется виртуальной машиной. CPython предоставляет высший уровень совместимости пакетов Python с модулями расширения, написанными на C[2].

Если вы пишете программу с открытым исходным кодом и хотите охватить наиболее широкую аудиторию, обратитесь к CPython. При использовании пакетов, функционирование которых зависит от расширений, написанных на C, CPython — ваш единственный вариант реализации.

Все версии языка Python созданы на языке C, поскольку CPython является базовой реализацией.

Stackless

Stackless Python (https://bitbucket.org/stackless-dev/stackless/wiki/Home) — это обычный вариант CPython (поэтому данная реализация будет работать со всеми библиотеками, которые может использовать CPython). Эта версия языка имеет патч, отвязывающий интерпретатор Python от стека вызовов, что позволяет изменять порядок выполнения кода. Stackless вводит концепцию *тасклетов*, которые могут оборачивать функции и превращать их в «микропотоки» (они могут быть сериа-

[1] Термин «базовая реализация» точно отражает определение языка. Его поведение указывает на то, как должны вести себя другие реализации.

[2] Модули расширения написаны на C, но могут использоваться и в Python.

лизованы на диск для последующего выполнения и планирования, по умолчанию выполняются по методу циклического перебора).

Библиотека greenlet (http://greenlet.readthedocs.org/) реализует такую же функциональность по смене стека для пользователей CPython. Большая ее часть также реализована в PyPy.

PyPy

PyPy (http://pypy.org/) — это интерпретатор Python, реализованный в ограниченном подмножестве статически типизированных языков Python (которое называется RPython), что позволяет выполнить оптимизацию. Интерпретатор предоставляет функциональность компиляции на лету и поддерживает несколько бэкендов, например C, язык CIL (Common Intermediate Language) (http://bit.ly/standard-ecma-335) и байт-код виртуальной машины Java (Java Virtual Machine, JVM).

PyPy нацеливается на максимальную совместимость с базовой реализацией CPython (http://speed.pypy.org/), при этом улучшая производительность. Если вы хотите повысить производительность вашего кода Python, стоит опробовать в деле PyPy. Согласно тестам производительности (бенчмаркам), в данный момент PyPy быстрее CPython более чем в пять раз. Он поддерживает Python 2.7, а PyPy3 (http://pypy.org/compat.html) нацелен на Python 3. Обе версии можно найти на странице загрузки PyPy (http://pypy.org/download.html).

Jython

Jython (http://www.jython.org/) — это реализация интерпретатора Python, компилирующая код Python в байт-код Java, который затем выполняется JVM. Дополнительно он может импортировать и использовать любой класс Java в качестве модуля Python.

Если вам нужно создать интерфейс для существующей базы кода Java (или же у вас есть другие причины писать код Python для JVM), Jython будет лучшим выбором.

Jython в данный момент поддерживает версии Python вплоть до Python 2.7 (http://bit.ly/jython-supports-27).

IronPython

IronPython (http://ironpython.net/) — это реализация Python для фреймворка .NET. Она может использовать библиотеки, написанные как на Python, так и с помощью .NET, а также предоставлять доступ к коду Python другим языкам фреймворка .NET.

Надстройка Python Tools for Visual Studio (http://ironpython.net/tools/) интегрирует IronPython непосредственно в среду разработки Visual Studio, что делает эту реализацию идеальным выбором для разработчиков, использующих Windows.

IronPython поддерживает Python 2.7 (http://ironpython.codeplex.com/releases/view/81726).

PythonNet

Python for .NET (http://pythonnet.github.io/) — это пакет, который предоставляет практически бесшовную интеграцию нативно установленного Python и общеязыковой среды выполнения .NET (Common Language Runtime (CLR)). Такой подход противоположен подходу, которым пользуется IronPython. Это означает, что PythonNet и IronPython дополняют друг друга, а не конкурируют.

В совокупности с Mono (http://www.mono-project.com/) PythonNet позволяет нативным установкам Python в операционных системах, отличающихся от Windows, например OS X и Linux, работать с фреймворком .NET. Реализация может быть без конфликтов запущена вместе с IronPython.

PythonNet поддерживает версии от Python 2.3 до Python 2.7. Инструкции по установке вы найдете на странице http://pythonnet.github.io/readme.html.

Skulpt

Skulpt (http://www.skulpt.org/) — это реализация Python на JavaScript. Для нее еще не была полностью портирована стандартная библиотека CPython. Библиотека включает в себя модули math, random, turtle, image, unittest, а также части модулей time, urllib, DOM и re. Предназначена для использования при обучении. Кроме того, есть возможность добавить собственные модули (http://bit.ly/skulpt-adding-module).

Стоит упомянуть примеры ее применения — Interactive Python (http://interactivepython.org/) и CodeSkulptor (http://www.codeskulptor.org/demos.html).

Skulpt поддерживает большую часть функциональности версий Python 2.7 и Python 3.3. Для получения более подробной информации смотрите страницу реализации Skulpt на GitHub (https://github.com/skulpt/skulpt).

MicroPython

MicroPython (https://micropython.org/) — это реализация Python 3, оптимизированная для запуска на микроконтроллере. Поддерживает 32-битные процессоры ARM, имеющие набор инструкций Thumb v2, такие как Cortex-M, широко используемые в дешевых микроконтроллерах. Включает в себя модули стандартной библиотеки Python (http://bit.ly/micropython-library), а также несколько

библиотек, характерных для MicroPython, которые отвечают за подробную информацию о плате и памяти, доступ к сетям, а также модифицированную версию ctypes, оптимизированную для уменьшения размера. Эта реализация не похожа на Raspberry Pi (https://www.raspberrypi.org/) — та поддерживала Debian или другую операционную систему, основанную на C, на которой установлен Python. Плата pyboard (https://micropython.org/store/#/store) использует MicroPython как свою операционную систему.

Здесь и далее мы будем использовать CPython в Unix-подобной системе, OS X или Windows.

Итак, перейдем к установке.

2 Правильная установка Python

В этой главе показан процесс установки CPython на платформах Mac OS X, Linux и Windows. Информация в разделах, посвященных инструментам упаковки (вроде Setuptools и pip), повторяется, поэтому можете сразу перейти к разделу, касающемуся вашей системы.

Если вы работаете в организации, которая рекомендует вам использовать коммерческий дистрибутив Python вроде Anaconda или Canopy, нужно следовать инструкциям от поставщика. Кроме того, для вас будет полезной информация раздела «Коммерческие дистрибутивы Python» в конце этой главы.

Если в вашей системе уже установлен Python, ни при каких обстоятельствах не позволяйте никому менять символьную ссылку на исполнительные файлы Python. Это может закончиться примерно так же плохо, как и декламация вогонской поэзии (https://en.wikipedia.org/wiki/Vogon#Poetry). (Подумайте о системном коде, зависящем от определенной версии Python, которая находится в конкретном месте…)

Установка Python на Mac OS X

Самая поздняя версия Mac OS X — El Capitan[1] — поставляется с собственной реализацией Python 2.7.

Вам *не нужно* устанавливать или конфигурировать что-то еще перед использованием Python. Но мы настоятельно рекомендуем установить Setuptools, pip и virtualenv до того, как вы начнете создавать приложения Python для применения в реальном мире (например, для внесения вклада в совместные проекты). В этом разделе вы

[1] На момент написания книги.

узнаете больше о том, как их устанавливать и использовать. В частности, вам всегда следует устанавливать Setuptools, поскольку это значительно упрощает применение других сторонних библиотек Python.

Та версия Python, которая поставляется вместе с OS X, отлично подходит для обучения, но не годится для совместной разработки. Эта версия может быть устаревшей по сравнению с официальной текущей версией, которая считается стабильной производственной версией[1]. Поэтому, если вы хотите использовать Python только для того, чтобы писать сценарии для себя, получать информацию с сайтов или обрабатывать данные, вам больше ничего не понадобится. Но если вы собираетесь вносить свой вклад в проекты с открытым исходным кодом или работать в команде с сотрудниками, которые могут пользоваться другими операционными системами (или планируете заниматься этим в будущем[2]), выбирайте версию CPython.

Перед тем как вы что-либо загрузите, прочтите следующие абзацы. До установки Python вам нужно установить GCC. Для этого загрузите Xcode (http://developer.apple.com/xcode/), упрощенную версию Command-Line Tools (https://developer.apple.com/downloads/) (вам понадобится учетная запись Apple) или даже версию пакета osx-gcc-installer (http://bit.ly/osx-gcc-installer-package).

Если вы уже установили Xcode, не устанавливайте osx-gcc-installer. Их совместная работа может привести к проблемам, которые трудно диагностировать.

Несмотря на то что OS X поставляется с большим количеством утилит Unix, разработчики, знакомые с системами Linux, обнаружат отсутствие одного ключевого компонента: подходящего менеджера пакетов. Эту брешь может заполнить инструмент Homebrew.

---

[1] У разных людей разные мнения на этот счет. Реализации OS X Python могут различаться. Они даже могут иметь разные библиотеки, характерные для OS X. Вы можете прочитать небольшую статью, автор которой критикует наши рекомендации, в блоге Stupid Python Ideas (http://bit.ly/sticking-with-apples-python). В ней поднимаются логичные вопросы о конфликтах имен, которые возникают, когда пользователь переключается с CPython 2.7 для OS X на CPython 2.7, независимый от конечной реализации. Если для вас это может стать проблемой, используйте виртуальную среду. Или хотя бы оставьте CPython 2.7 для OS X в том же месте, чтобы система четко работала, установите стандартный Python 2.7, реализованный с помощью CPython, измените путь и никогда не пользуйтесь версией OS X. Далее все будет работать как часы, включая продукты, которым нужна версия Python для OS X.

[2] Честно говоря, лучший вариант — с самого начала использовать либо Python 3, либо виртуальные среды, куда вы не будете устанавливать ничего, кроме virtualenv и, возможно, virtualenvwrapper в соответствии с советами Хинека Шлавака (Hynek Schlawack) (https://hynek.me/articles/virtualenv-lives/).

Чтобы установить Homebrew, откройте Terminal или ваш любимый эмулятор консоли для OS X и запустите следующий код:

```
$ BREW_URI=https://raw.githubusercontent.com/Homebrew/install/master/install
$ ruby -e "$(curl -fsSL ${BREW_URI})"
```

Сценарий объяснит, какие изменения он собирается внести, и предупредит вас перед началом процесса. Как только вы установите Homebrew, добавьте каталог `Homebrew` в начало вашей переменной среды (окружения) `PATH`[1]. Вы можете сделать это, добавив следующую строку в нижнюю часть файла `~/.profile`:

```
export PATH=/usr/local/bin:/usr/local/sbin:$PATH
```

Для установки Python запустите в консоли следующую команду:

```
$ brew install python3
```

Для Python 2:

```
$ brew install python
```

По умолчанию Python будет установлен в каталог `/usr/local/Cellar/python3/` или `/usr/local/Cellar/python/` с символьными ссылками[2] для интерпретатора по адресу `/usr/local/python3` или `/usr/local/python`. Если вы планируете использовать параметр `--user` для команды `pip install`, то придется обойти баг, который касается disutils и конфигурации Homebrew. Мы рекомендуем работать с виртуальными средами, описанными в подразделе «virtualenv» далее.

Setuptools и pip

Инструмент Homebrew установит Setuptools и pip. Исполняемые файлы, установленные с помощью pip, будут соотнесены для pip3, если вы используете Python 3, или для pip, если вы применяете Python 2.

С помощью Setuptools вы можете загрузить и установить любое совместимое[3] ПО для Python по сети (обычно по Интернету), выполнив всего одну команду

---

1 Это гарантирует, что версия Python, которую вы собираетесь использовать, будет той, которую только что установил Homebrew (оставил оригинальную версию Python нетронутой).

2 Символьная ссылка — это указатель на реальное местоположение файла. Вы можете проверить, куда ведет ссылка, набрав, например, ls -l /usr/local/bin/python3 в командной строке.

3 Пакеты, которые минимально совместимы с Setuptools, предоставляют достаточно информации для библиотеки, чтобы она могла идентифицировать и получить все зависимые пакеты. Для более подробной информации обратитесь к документации Packaging and Distributing Python Projects (https://packaging.python.org/en/latest/distributing.html), PEP 302 (https://www.python.org/dev/peps/pep-0302/) и PEP 241 (https://www.python.org/dev/peps/pep-0241/).

(`easy_install`). Это также позволит вам добавить возможность устанавливать софт по сети для ваших собственных программ, написанных на Python, не затратив много усилий.

Команда `pip` для pip и команда `easy_install` для Setuptools — средства для установки и управления пакетами Python. Первую использовать предпочтительнее, поскольку она тоже может удалять пакеты, ее сообщения об ошибке более понятны, а частичные установки пакетов невозможны (если процесс даст сбой, все его результаты будут отменены).

Более детальное сравнение приводится в статье *pip vs easy_install* (http://bit.ly/pip-vs-easy-install) в руководстве Python Packaging User (https://packaging.python.org/) (сюда следует обращаться всякий раз, когда вам требуется актуальная информация о пакетах).

Для улучшения установленной версии pip введите следующий код в консоли оболочки:

```
$ pip install --upgrade pip
```

virtualenv

Инструмент virtualenv создает изолированные среды Python — каталог, содержащий все исполняемые файлы, которые будут использовать пакеты, что могут понадобиться проекту, написанному на Python.

Некоторые считают, что хорошим тоном является установка только лишь virtualend и Setuptools и дальнейшая работа *исключительно* в виртуальных средах[1].

Для того чтобы установить virtualenv с помощью pip, запустите `pip` в командной строке консоли оболочки:

```
$ pip3 install virtualenv
```

Для Python 2:

```
$ pip install virtualenv
```

Как только вы окажетесь в виртуальной среде, всегда сможете использовать команду `pip` независимо от того, работаете вы с Python 2 или Python 3 (именно это мы и будем делать в остальной части руководства). В подразделе «Виртуальные среды» раздела «Инструменты изоляции» главы 3 использование и мотивация описываются более подробно.

---

[1] Сторонники такого подхода говорят, что это единственный способ гарантировать, что ничто не сможет переписать старую версию существующей библиотеки и тем самым навредить другому коду ОС, зависимому от версии библиотеки.

Установка Python на Linux

Ubuntu выпускается только с предустановленной версией Python 3 (версия Python 2 доступна по команде `apt-get`), начиная с версии Wily Werewolf (Ubuntu 15.10). Все подробности вы можете узнать на странице Python для Ubuntu (https://wiki.ubuntu.com/Python). Релиз Fedora 23 — первый, в котором доступен только Python 3 (обе версии — Python 2.7 и Python 3 — доступны в версиях 20–22), версия Python 2.7 будет доступна только благодаря менеджеру пакетов.

Большая часть параллельных установок Python 2 и Python 3 создает символьную ссылку на интерпретаторы Python 2 и Python 3. Если вы решите использовать Python 2, текущей рекомендацией для Unix-подобных систем (см. Python Enhancement Proposal [PEP 394] (https://www.python.org/dev/peps/pep-0394/)) является необходимость явно указывать python2 (например, `#!/usr/bin/env python2` в первой строке файла), не полагаясь на то, что среда сама все сделает.

Несмотря на то что этого нет в PEP 394, теперь принято использовать pip2 и pip3, чтобы указывать на соответствующие установщики пакетов.

Setuptools и pip

Хотя pip доступен в вашей системе благодаря установщику пакетов, для того чтобы гарантировать, что вы будете работать с самой новой версией, выполните следующие шаги.

1. Для начала загрузите `get-pip.py`[1] (https://bootstrap.pypa.io/get-pip.py).
2. Далее откройте оболочку, измените каталоги так, чтобы они указывали на то же место, что и `get-pip.py`, и введите следующий код:

```
$ wget https://bootstrap.pypa.io/get-pip.py
$ sudo python3 get-pip.py
```

Для Python 2:

```
$ wget https://bootstrap.pypa.io/get-pip.py
$ sudo python get-pip.py
```

Эти команды также установят Setuptools.

С помощью команды `easy_install`, которая доступна благодаря Setuptools, вы можете загрузить и установить любое совместимое[2] ПО для Python по сети (обычно

---

[1] Для получения более подробной информации обратитесь к инструкции по установке pip (https://pip.pypa.io/en/latest/installing.html).

[2] Пакеты, которые минимально совместимы с Setuptools, предоставляют достаточно информации для библиотеки, чтобы она могла идентифицировать и получить все зависимые пакеты. За более подробной информацией обратитесь к документации Packaging and Distributing Python Projects, PEP 302 и PEP 241.

по Интернету). Это также позволит вам добавить возможность устанавливать ПО по сети для ваших собственных программ, написанных на Python, не затратив много усилий.

Команда `pip` позволяет легко устанавливать пакеты Python и управлять ими. Рекомендуется использовать именно этот инструмент вместо `easy_install`, поскольку эта команда также может удалять пакеты, ее сообщения об ошибке более понятны, а частичные установки пакетов невозможны (если процесс даст сбой, все его результаты будут отменены).

Инструменты разработки

Практически каждый пользователь Python в какой-то момент захочет обратиться к библиотекам, которые зависят от расширений, написанных на языке C. Возможно, ваш менеджер пакетов будет иметь заранее собранные библиотеки, поэтому вы можете сначала проверить это (с помощью команд `yum search` или `apt-cache search`). Более новый формат *wheels* (http://pythonwheels.com/) (заранее скомпилированные характерные для платформы бинарные файлы) дает возможность получить бинарные файлы непосредственно из PyPI с помощью команды `pip`. Если вы планируете в будущем создавать расширения, написанные на C, или если люди, поддерживающие вашу библиотеку, не создали решения с помощью wheels для вашей платформы, вам понадобятся инструменты разработки для Python: разнообразные библиотеки, написанные на C, команда `make` и компилятор GCC. Перечислим полезные пакеты, которые работают с библиотеками, написанными на C.

Пакеты для работы с конкуренцией:

- библиотека для работы с потоками threading (https://docs.python.org/3/library/threading.html);
- библиотека для обработки событий (Python 3.4+) asyncio (https://docs.python.org/3/library/asyncio.html);
- библиотека, основанная на сопрограммах, curio (https://curio.readthedocs.org/);
- библиотека для работы с сетями, основанная на сопрограммах, gevent (http://www.gevent.org/);
- управляемая событиями библиотека для работы с сетями Twisted (https://twistedmatrix.com/).

Научный анализ:

- библиотека для работы с линейной алгеброй NumPy (http://www.numpy.org/);
- набор инструментов для работы с числами SciPy (http://www.scipy.org/);
- библиотека для работы с машинным обучением scikit-learn (http://scikit-learn.org/);
- библиотека для построения графиков Matplotlib (http://matplotlib.org/).

Интерфейс для работы с данными/базой данных:

- интерфейс для формата HDF5 h5py (http://www.h5py.org/);
- адаптер для базы данных PostgreSQL Psycopg (http://initd.org/psycopg/);
- абстракция базы данных и объектно-ориентированный менеджер памяти (mapper) SQLAlchemy (http://www.sqlalchemy.org/).

В Ubuntu в консоли оболочки введите следующий код:

```
$ sudo apt-get update --fix-missing
$ sudo apt-get install python3-dev  # Для Python 3
$ sudo apt-get install python-dev   # Для Python 2
```

В Fedora в консоли оболочки введите такой код:

```
$ sudo yum update
$ sudo yum install gcc
$ sudo yum install python3-devel   # Для Python 3
$ sudo yum install python2-devel   # Для Python 2
```

С помощью команды `pip3 install --user` *`желаемый_пакет`* вы сможете выполнить сборку для инструментов, которые должны быть скомпилированы. (Или `pip install --user` *`желаемый_пакет`* для Python 2.) Вам также потребуется установить сам инструмент (чтобы узнать, как это делается, обратитесь к документации по установке HDF5 (https://www.hdfgroup.org/HDF5/release/obtain5.html)). Для PostgreSQL в Ubuntu вам необходимо ввести следующий код в консоли оболочки:

```
$ sudo apt-get install libpq-dev
```

Для Fedora:

```
$ sudo yum install postgresql-devel
```

virtualenv

Команда `virtualenv` доступна после установки пакета virtualenv (https://pypi.python.org/pypi/virtualenv), который создает изолированные среды Python. Она создает каталог, содержащий все необходимые исполняемые файлы пакетов, которые могут понадобиться для проекта, написанного на Python.

Для того чтобы установить virtualenv с помощью менеджера пакетов Ubuntu, введите следующий код:

```
$ sudo apt-get install python-virtualenv
```

Для Fedora:

```
$ sudo yum install python-virtualenv
```

Вы можете установить пакет с помощью команды `pip`. Запустите менеджер в командной строке консоли оболочки и добавьте параметр `--user`, чтобы установить пакет локально для себя (не выполняя установку для всей системы):

```
$ pip3 install --user virtualenv
```

Для Python 2:

```
$ sudo pip install --user virtualenv
```

Как только вы окажетесь в виртуальной среде, всегда сможете использовать команду `pip` независимо от того, работаете вы с Python 2 или Python 3 (что мы и будем делать на протяжении остальной части этого руководства).

Установка Python на Windows

Пользователям Windows приходится труднее остальных, поскольку в этой операционной системе сложнее выполнять компиляцию и многие библиотеки Python втайне используют расширения, написанные на C.

Благодаря формату wheels бинарные файлы можно загрузить из PyPI с помощью `pip` (если они существуют), поэтому работать с Python стало несколько проще.

У вас есть два пути: использовать коммерческий дистрибутив (они рассматриваются в разделе «Коммерческие дистрибутивы Python» далее) или CPython. Работать с дистрибутивом Anaconda гораздо проще, особенно если вы собираетесь заниматься научными расчетами. Практически каждый разработчик, применяющий Python для Windows (кроме тех, кто самостоятельно разрабатывает библиотеки для Python, основанные на C), порекомендует Anaconda. Но если вы разбираетесь в процессах компилирования и связывания, если хотите вносить свой вклад в проекты, которые используют код на C, или не желаете выбирать коммерческий дистрибутив (вам нужны только бесплатные функции), рассмотрите возможность установки CPython[1] (когда требуется интегрировать Python во фреймворк .NET). Однако, если вы только начинаете осваивать Python, знакомство с этим интерпретатором, скорее всего, лучше отложить на будущее. На протяжении этой книги мы рассказываем о CPython.

Со временем все больше пакетов, содержащих библиотеки, написанные на C, будут поддерживать формат wheels для PyPI. Их можно будет получить, воспользовавшись командой `pip`. Проблемы могут возникнуть, если у зависимости для необходимой вам библиотеки нет решения, написанного в соответствии с wheel. Это одна из причин, почему вы можете предпочесть коммерческие дистрибутивы Python вроде Anaconda.

[1] Или IronPython (рассматривается в подразделе «IronPython» раздела «Реализации» главы 1).

Вам следует использовать CPython, если вы работаете под Windows и при этом:

- вам не нужны библиотеки Python, которые зависят от расширений, написанных на C;
- у вас есть компилятор для Visual C++ (не тот, что распространяется бесплатно);
- можете настроить MinGW;
- можете загрузить бинарные файлы вручную[1], а затем установить их с помощью команды `pip install`.

Если вы планируете использовать Python в качестве замены R или MATLAB или же хотите быстро включиться в работу и установить CPython позже при необходимости (в разделе «Коммерческие дистрибутивы Python» далее вы сможете найти несколько советов), выбирайте Anaconda[2].

Если желаете работать в IDE (integrated development environments — интегрированная среда разработки), чей интерфейс в основном графический («указать и щелкнуть»), или если Python — ваш первый язык программирования и вам предстоит окунуться в первую установленную IDE, используйте Canopy.

Когда вся ваша команда уже применяет один из представленных здесь вариантов, вам следует действовать так же.

Чтобы установить стандартную реализацию CPython для Windows, для начала понадобится загрузить последнюю версию Python 3 (https://www.python.org/ftp/python/3.5.0/python-3.5.0.exe) или Python 2.7 (https://www.python.org/ftp/python/2.7.10/python-2.7.10.msi) с официального сайта. Если вы хотите быть полностью уверены в том, что устанавливаете последнюю версию (либо же знаете, что вам необходим 64-битный установщик[3]), можете воспользоваться ресурсом Python Releases for Windows (https://www.python.org/downloads/windows/) (там найдете необходимый вам релиз).

---

[1] Вы должны знать хотя бы версию Python, которую планируете использовать, а также требуемую разрядность. Мы рекомендуем 32-битную версию, поскольку все сторонние DLL имеют 32-битную версию, но не все — 64-битную. Самое популярное место, где вы можете найти скомпилированные бинарные файлы, — сайт ресурсов Кристофа Голка (Christoph Gohlke) (http://www.lfd.uci.edu/~gohlke/pythonlibs/). Что касается scikit-learn, Карл Клеффнер (Carl Kleffner) выполняет сборку бинарных файлов с помощью MinGW (https://pypi.anaconda.org/carlkl/simple/) для подготовки к итоговому релизу для PyPI.

[2] В Anaconda больше бесплатных функций, он поставляется вместе со Spyder — более качественной IDE. Если вы используете Anaconda, для вас может оказаться полезным предметный указатель бесплатного пакета Anaconda (https://repo.continuum.io/pkgs/) и пакета Canopy (https://www.enthought.com/products/canopy/package-index/).

[3] Это означает, что вы на 100 % уверены, что все необходимые DLL и драйверы имеют 64-битную версию.

Версия для Windows включает пакет MSI. Этот формат позволяет администраторам Windows автоматизировать установку с помощью стандартных инструментов. Для установки пакета вручную дважды щелкните на файле.

По умолчанию Python устанавливается в каталог, в название которого встроен номер версии (например, версия Python 3.5 будет установлена в каталог `C:\Python35\`), поэтому у вас может быть несколько версий Python на одной системе и при этом не будет конфликтов. Конечно же, по умолчанию можно использовать всего один интерпретатор. Установщик не изменяет переменную среды `PATH`[1]. Все записи разделены точкой с запятой автоматически, поэтому вы всегда сможете самостоятельно выбирать, какую именно копию Python следует запустить.

Вводить полный путь к интерпретатору Python каждый раз утомительно, поэтому добавьте каталоги, в которых хранится версия Python, используемая по умолчанию, в переменную среды `PATH`. Если эта версия находится в каталоге `C:\Python35\`, укажите следующий код в переменной `PATH`:

```
C:\Python35;C:\Python35\Scripts\
```

Вы можете сделать это, запустив в PowerShell1[2] следующие команды:

```
PS C:\> [Environment]::SetEnvironmentVariable(
    "Path",
    "$env:Path;C:\Python35\;C:\Python35\Scripts\",
    "User")
```

Второй каталог (`Scripts`) получает файлы команд, когда устанавливаются определенные пакеты (это очень полезное дополнение). Вам не потребуется устанавливать или конфигурировать что-то еще для применения Python.

С другой стороны, мы настоятельно рекомендуем установить Setuptools, pip и virtualenv перед тем как запускать приложения Python в работу (то есть для работы с совместным проектом). Далее в текущем разделе вы получите более подробную информацию об этих инструментах. В частности, всегда нужно устанавливать Setuptools, поскольку это значительно упростит использование других сторонних библиотек для Python.

---

[1] В переменной среды PATH перечислены все возможные каталоги, в которых операционная система будет искать исполняемые программы (например, Python и его сценарии вроде pip).

[2] Windows PowerShell предоставляет язык сценариев для командной строки, который напоминает оболочки Unix — пользователи Unix могут работать с ним, не читая руководство, но при этом он имеет функциональность, которую можно применять только в Windows. Создан на основе фреймворка .NET. Более подробную информацию ищите в статье «Using Windows PowerShell от компании Microsoft» (http://bit.ly/using-windows-powershell).

Setuptools и pip

MSI устанавливают Setuptools и pip вместе с Python, поэтому, если вы выполняете все рекомендации из этой книги и только что произвели установку, они у вас уже есть. В противном случае лучший способ их получить при работе с версией 2.7 — выполнить обновление до самой последней версии 15 (установщик спросит у вас, можно ли перезаписать существующую версию, — вам следует согласиться; релизы, имеющие одинаковый вспомогательный номер версии, имеют и обратную совместимость). Для Python 3 (версии 3.3 и младше) загрузите сценарий `get-pip.py`[1] (https://bootstrap.pypa.io/get-pip.py) и запустите его. Откройте оболочку, измените каталог на тот, в котором находится `get-pip.py`, и введите следующий код:

```
PS C:\> python get-pip.py
```

С помощью Setuptools вы можете по сети (обычно по Интернету) загрузить и установить любое совместимое[2] ПО для Python, введя одну команду (`easy_install`). Это также позволит добавить возможность устанавливать софт по сети для ваших собственных программ, написанных с помощью Python, не затратив много усилий.

Команда `pip` для pip и команда `easy_install` для Setuptools являются инструментами для установки и управления пакетами Python. Первую команду использовать предпочтительнее, поскольку она также может удалять пакеты, ее сообщения об ошибке более понятны, а частичные установки пакетов невозможны (если процесс установки даст сбой, все его результаты будут отменены).

virtualenv

Команда `virtualenv` (http://pypi.python.org/pypi/virtualenv) позволяет создавать изолированные среды Python. Она создает каталог, содержащий все необходимые исполняемые файлы для использования пакетов, которые могут понадобиться для проекта, написанного на Python. Когда вы активизируете среду с помощью команды в новом каталоге, она добавит его название в конец строки, представляющей собой переменную среды `PATH` — версия Python в новом каталоге будут обнаружена в первую очередь, в результате чего будут задействованы пакеты в его подкаталогах.

---

[1] Для получения более подробной информации обратитесь к инструкциям по установке pip (https://pip.pypa.io/en/latest/installing.html).

[2] Пакеты, которые минимально совместимы с Setuptools, предоставляют достаточно информации для библиотеки, чтобы она могла идентифицировать и получить все зависимые пакеты. Более подробные сведения ищите в документации Packaging and Distributing Python Projects (https://packaging.python.org/en/latest/distributing.html), PEP 302 (https://www.python.org/dev/peps/pep-0302/) и PEP 241 (https://www.python.org/dev/peps/pep-0241/).

Для того чтобы установить virtualenv с помощью `pip`, введите эту команду в командной строке консоли PowerShell:

```
PS C:\> pip install virtualenv
```

В OS X и Linux (поскольку Python предустанавливается для использования системой или сторонним ПО) необходимо явно разграничивать версии pip для Python 2 и Python 3. В Windows вам не нужно этого делать, поэтому, когда мы говорим `pip3`, имеем в виду `pip` для пользователей Windows. Независимо от ОС, как только вы попадаете в виртуальную среду, всегда можете использовать команду pip — неважно, работаете вы с Python 2 или Python 3 (это мы и будем делать на протяжении остальной части книги).

Коммерческие дистрибутивы Python

Ваш отдел IT или преподаватель могут попросить вас установить коммерческий дистрибутив Python. Это необходимо, чтобы упростить работу, которую должна выполнить организация, и поддерживать стабильную среду для нескольких пользователей. Все перечисленные здесь дистрибутивы предоставляют реализацию Python, написанную на C (CPython).

Научный редактор первого черновика этой главы сказал, что мы серьезно недооцениваем неудобства, которые большинству пользователей доставляет обычная версия CPython на Windows: несмотря на существование формата wheels, компилирование и/или связывание внешних библиотек, написанных на C, представляет трудность для всех, кроме опытных разработчиков. Мы предпочитаем CPython, но, если вы собираетесь пользоваться библиотеками или пакетами (а не создавать их или добавлять в них что-то свое), вам следует загрузить коммерческий дистрибутив и просто начать работать (это особенно важно, если вы работаете в Windows). Когда захотите внести свой вклад в проекты с открытым исходным кодом, сможете установить обычный дистрибутив CPython.

Вернуться к оригинальной версии Python будет проще, если вы не станете изменять настройки по умолчанию при установке версий Python от сторонних поставщиков.

Кратко опишем коммерческие дистрибутивы.

- *Intel Distribution*. Предоставляет удобный и бесплатный доступ к высокоскоростной реализации Python (https://software.intel.com/en-us/python-distribution). Основной прирост производительности отмечается благодаря связыванию пакетов Python с нативными библиотеками вроде Intel Math Kernel Library (MKL), улучшению работы с потоками, а также благодаря библиотеке Intel Threading

Building Blocks (TBB) (http://bit.ly/intel-tbb-for-python). Для управления пакетами используется conda от Continuum, но подойдет и pip. Дистрибутив можно загрузить самостоятельно либо установить с сайта https://anaconda.org/ в среде conda[1] (http://bit.ly/intel-python-beta).

Дистрибутив предоставляет стек SciPy и другие распространенные библиотеки, перечисленные в сопроводительных документах (в формате PDF) (http://bit.ly/intel-python-release-notes). Пользователям Intel Parallel Studio XE доступна коммерческая поддержка, а все остальные могут общаться на форумах. Позволяет вам без особого труда обращаться к научным библиотекам, в остальном ничем не отличается от обычного дистрибутива Python.

- *Anaconda от Continuum Analytics.* Дистрибутив Python от Continuum Analytics (https://www.continuum.io/downloads) выпущен под лицензией BSD и предоставляет множество заранее скомпилированных научных и математических бинарных файлов в своем каталоге бесплатных пакетов (https://repo.continuum.io/pkgs/). Он использует не pip, а другой менеджер пакетов (conda), который также управляет виртуальными средами, но действует скорее как Buildout (рассматривается в подразделе «Buildout» раздела «Инструменты изоляции» главы 3), а не как virtualenv (управляет библиотеками и другими внешними зависимостями для пользователя). Форматы пакетов несовместимы, поэтому вы не сможете вызвать один установщик из каталога пакетов другого.

  Дистрибутив Anaconda поставляется со стеком SciPy и другими инструментами. Anaconda имеет отличную лицензию и максимум бесплатной функциональности. Если вам комфортно работать с командной строкой и нравится R или Scala (идут в комплекте), то лучше выбрать коммерческий дистрибутив. Если подобная функциональность не требуется, используйте вместо него miniconda (http://conda.pydata.org/miniconda.html). Пользователи получают разнообразные компенсации (связанные с лицензией на ПО с открытым исходным кодом, а также проясняющие, кто, чем и когда может пользоваться и на кого и в каких случаях подадут в суд), коммерческую поддержку и дополнительные библиотеки Python.

- *ActivePython от ActiveState.* Дистрибутив от ActiveState (http://www.activestate.com/downloads) выпущен под лицензией ActiveState Community License и бесплатен только во время пробного периода, потом понадобится лицензия. ActiveState предоставляет решения для Perl и Tcl. Основной плюс этого дистрибутива — его широкие возможности по выплате компенсаций (связанных с лицензией на ПО с открытым исходным кодом) для более чем 7000 пакетов, расположенных в его каталоге пакетов (https://code.activestate.com/pypm/) (их можно получить с помощью инструмента pypm, заменяющего pip).

---

[1] Intel и Anaconda — партнеры (http://bit.ly/announce-anaconda-intel), и все специализированные пакеты компании Intel (https://anaconda.org/intel) доступны только при использовании conda. Однако вы всегда можете выполнить команду conda install pip и применить pip (или pip install conda и использовать conda).

- *Canopy от Enthought*. Дистрибутив от Enthought (https://store.enthought.com/downloads/) выпущен под лицензией Canopy Software License, включает в себя менеджер пакетов enpkg, который используется вместо pip для связи с каталогом пакетов Canopy (http://bit.ly/enthought-canopy).

  Enthought предоставляет бесплатные академические лицензии студентам и работникам учреждений образования. Отличительными особенностями дистрибутива от Enthought являются графические инструменты для взаимодействия с Python, которые включают собственную IDE, напоминающую MATLAB, а также графический менеджер пакетов, графический отладчик и графический инструмент для взаимодействия с данными. Как и у других коммерческих дистрибутивов, в нем предусмотрены механизм возмещения ущерба и коммерческая поддержка, а также дополнительные пакеты для покупателей.

3 Ваша среда разработки

В этой главе приведен обзор текстовых редакторов, интегрированных сред разработки и других инструментов разработки, популярных в настоящий момент.

Мы предпочитаем использовать Sublime Text в качестве редактора и PyCharm/IntelliJ IDEA (оба рассматриваются в следующем разделе) в качестве IDE, но при этом понимаем, что лучший вариант зависит от решаемых вами задач и используемых языков программирования. В этой главе перечислены самые популярные приложения и их достоинства/недостатки.

Для Python не нужны инструменты сборки вроде Make, Java's Ant или Maven, поскольку это интерпретируемый, а не компилируемый язык[1], поэтому мы не будем рассматривать эти вопросы. Но в главе 6 опишем, как использовать Setuptools для упаковки проектов и Sphinx для сборки документации.

Мы также не будем рассматривать системы контроля версий, поскольку они не зависят от языка, но программисты, которые поддерживают реализацию Python, написанную на C (ссылочную), не так уж давно перешли с Mercurial на Git (см. PEP 512 (https://www.python.org/dev/peps/pep-0512/)). Оригинальное обоснование использования Mercurial в PEP 374 (https://www.python.org/dev/peps/pep-0374/) небольшое, зато представлено сравнение четырех вариантов, популярных в настоящее время: Subversion, Bazaar, Git и Mercurial.

Завершается глава кратким обзором современных способов управления интерпретаторами для воссоздания разных ситуаций, которые могут возникнуть при развертывании, на этапе кодирования.

---

[1] Если вам понадобится создавать расширения для Python с помощью языка C, обратитесь к статье «Расширяем Python с помощью C или C++» (https://docs.python.org/3/extending/extending.html). Для получения более подробной информации см. главу 15 книги «Python Cookbook» (http://bit.ly/python-cookbook).

Текстовые редакторы

Для написания кода Python подойдет любая программа, которая позволяет редактировать текст, однако выбор правильного редактора сэкономит вам несколько часов в неделю. Все текстовые редакторы, перечисленные в этом разделе, поддерживают подсветку синтаксиса и могут быть расширены с помощью надстроек таким образом, чтобы выполнять статическую проверку кода (с помощью средств контроля качества кода) и делать отладку.

В табл. 3.1 перечислены текстовые редакторы, которые нам нравятся (по убыванию предпочтения), и объясняется, почему разработчику следует выбрать именно этот редактор. Далее в главе кратко рассматривается каждый из них. В «Википедии» по адресу https://en.wikipedia.org/wiki/Comparison_of_text_editors приведена таблица, в которой детально сравниваются текстовые редакторы (поможет тем, кто ищет определенную функциональность).

Таблица 3.1. Первый взгляд на текстовые редакторы

Инструмент	Доступность	Причина использовать
Sublime Text	• Открытый API/бесплатный пробный период. • OS X, Linux, Windows	• Быстро работает и задействует небольшой объем памяти. • Способен работать с крупными файлами (> 2 Гбайт). • Расширения написаны на Python
Vim	• ПО с открытым исходным кодом/можно вносить пожертвования. • OS X, Linux, Windows, Unix	• Вам нравится Vi/Vim. • Предустановлен (во всяком случае Vi) на каждой ОС кроме Windows. • Может быть консольным приложением
Emacs	• ПО с открытым исходным кодом/можно вносить пожертвования. • OS X, Linux, Windows, Unix	• Вам нравится Emacs. • Расширения написаны на Lisp. • Может быть консольным приложением
TextMate	• ПО с открытым исходным кодом/нужна лицензия. • Только для OS X	• Отличный пользовательский интерфейс. • Практически все интерфейсы (статическая проверка кода/отладка/тестирование) предустановлены. • Хорошие инструменты от Apple, например интерфейс для xcodebuild (его можно найти в Xcode bundle)

Продолжение ⇗

Таблица 3.1 *(продолжение)*

Инструмент	Доступность	Причина использовать
Atom	• ПО с открытым исходным кодом/бесплатный. • OS X, Linux, Windows	• Расширения написаны на JavaScript/HTML/CSS. • Хорошая интеграция с GitHub
Code	• Открытый API (в будущем)/бесплатный. • OS X, Linux, Windows (но Visual Studio, соответствующая IDE, работает только в Windows)	• IntelliSense (автозаполнение кода), предоставляемое VisualStudio. • Хорошо подходит для разработчиков на Windows, поскольку поддерживает .Net, C# и F# Примечание. Имеется недостаток: пока нельзя расширять (в будущем будет исправлен)

Sublime Text

Мы советуем использовать текстовый редактор Sublime Text (http://www.sublimetext.com/) для написания кода, разметки и просто текста. В первую очередь его рекомендуют из-за скорости; далее в списке достоинств идет количество доступных пакетов (3000+).

Редактор Sublime Text был выпущен в 2008 году Jon Skinner. Написан на Python, позволяет редактировать код Python и использовать этот язык, работая со своим API для расширения пакетов. Функция Projects (Проекты) позволяет пользователю добавлять/удалять файлы или каталоги (потом их можно найти с помощью функции Goto Anything (Перейти), которая определяет места внутри проекта, содержащие искомые элементы).

Для того чтобы получить доступ к репозиторию пакетов Sublime Text, вам понадобится PackageControl (https://packagecontrol.io/installation). Популярные пакеты включают SublimeLinter, интерфейс для работы с выбранными и установленными пользователем статическими проверками кода, Emmett для работы со сниппетами[1] для веб-разработчиков и Sublime SFTP для удаленного редактирования с помощью FTP.

Редактор Anaconda (http://damnwidget.github.io/anaconda/) (не связан с одноименным коммерческим дистрибутивом Python), выпущенный в 2013 году, превращает Sublime практически в полноценный IDE, дополняя его статическими проверками кода, проверками строк документации, инструментом для запуска тестов и возможностью найти определение выделенных объектов или места, где они используются.

[1] Сниппеты — это фрагменты кода, которые часто повторяются, вроде стилей CSS или определений классов. Вы можете использовать для них автозаполнение, если введете несколько символов, а затем нажмете клавишу Tab.

Vim

Vim — консольный текстовый редактор (иногда имеет графический интерфейс), в котором для редактирования используются горячие клавиши вместо меню и значков. Выпущен в 1991 году Брамом Муленааром (Bram Moolenaar); его предшественник Vi — в 1976-м Биллом Джоем (Bill Joy). Оба редактора написаны на C.

Vim можно расширить с помощью vimscript — простого языка для написания сценариев. Можно использовать другие языки. Для включения возможности написания сценариев на Python установите следующие флаги конфигурирования при сборке исходного кода, написанного на C: `--enable-pythoninterp` и/или `--enable-python3interp`. После этого можете начать сборку исходников. Чтобы узнать, какие версии Python вам доступны, введите `:echo has("python")` или `:echo has("python3")`. Результатом будет `1`, если выражение верно, и `0`, если нет.

Vi (и зачастую Vim) доступен сразу после установки практически в любых операционных системах, кроме Windows (для нее существует установщик Vim — http://www.vim.org/download.php#pc). Основные сочетания клавиш, используемые в Vi, доступны в большинстве других редакторов и IDE.

Если хотите работать в крупной компании на должности, связанной с IT, вам необходимо уметь работать с Vi[1]. У Vim гораздо больше функций, чем у Vi, но он похож на предшественника настолько, что пользователь Vim справится с Vi.

Если вы пишете код только на Python, задайте настройки по умолчанию для отступов и переноса строк, чтобы значения были совместимы с PEP 8 (https://www.python.org/dev/peps/pep-0008). Для этого создайте в домашнем каталоге[2] файл с именем `.vimrc` и введите в него следующий код:

```
set textwidth=79  " строки длиннее 79 символов будут разбиваться
set shiftwidth=4  " операция >> сдвигает на 4 позиции вправо;
                  " << сдвигает на 4 позиции влево
set tabstop=4     " табуляция имеет длину 4 позиции
set expandtab     " использовать пробелы при табуляции
set softtabstop=4 " добавление/удаление четырех пробелов при нажатии
                  " клавиш TAB/BACKSPACE
set shiftround    " округлять длину отступа до значения,
                  " кратного значению параметра 'shiftwidth'
set autoindent    " задавать для новой строки такой же отступ,
                  " что и для предыдущей
```

[1] Просто откройте редактор, введя в командной строке vi (или vim) и нажав клавишу Enter, потом введите :help и нажмите Enter для прохождения обучения.

[2] Для того чтобы найти домашний каталог в Windows, откройте Vim и введите команду :echo $HOME.

С помощью этих настроек символ перехода на новую строку будет добавляться после каждых 79 символов в строке. Отступы настроены таким образом, что при каждом нажатии табуляции будет добавлено четыре пробела. Если вы находитесь внутри выражения, имеющего отступ, следующая строка будет иметь отступ такого же уровня.

Существует также надстройка для синтаксиса (называется python.vim, http://bit.ly/python-vim), которая улучшает файл синтаксиса, включенный в Vim 6.1, а также небольшая надстройка SuperTab (http://bit.ly/supertab-vim), позволяющая использовать более удобное автозаполнение кода путем нажатия Tab (или любой другой клавиши, указанной в настройках).

Если вы используете Vim для других языков, обратите внимание на удобную надстройку indent (http://bit.ly/indent-vim), которая работает с настройками отступов в файлах исходного кода Python.

Эти надстройки предоставляют основную среду для разработки на Python. Если ваша копия Vim скомпилирована с параметром `+python` (вариант по умолчанию для версии Vim 7 и выше), вы также можете использовать надстройку vim-flake8 (https://github.com/nvie/vim-flake8) для выполнения статических проверок кода внутри редактора. Эта надстройка предоставляет функцию `Flake8`, которая запускает pep8 (http://pypi.python.org/pypi/pep8/) и Pyflakes (http://pypi.python.org/pypi/pyflakes/). Надстройка отобразит ошибки в нижней части экрана и предложит простой способ переключиться на соответствующую строку.

Если вам удобно, можете заставить Vim вызывать `Flake8` каждый раз, когда вы сохраняете файл с кодом Python, добавив следующую строку в файл `.vimrc`:

```
autocmd BufWritePost *.py call Flake8()
```

Или, если вы уже пользуетесь плагином syntastic (https://github.com/scrooloose/syntastic), то можете настроить его так, чтобы он запускал Pyflakes при записи и показывал ошибки и предупреждения в окне quickfix. Рассмотрим пример конфигурации, которая позволяет этого достичь, а также выдавать сообщения о состоянии и предупреждения в строке состояния:

```
set statusline+=%#warningmsg#
set statusline+=%{SyntasticStatuslineFlag()}
set statusline+=%*
let g:syntastic_auto_loc_list=1
let g:syntastic_loc_list_height=5
```

Python-mode. Python-mode (https://github.com/klen/python-mode) — сложное решение для работы с кодом Python в Vim. Если вам понравятся представленные здесь функции, стоит им воспользоваться (но имейте в виду, что это немного замедлит запуск Vim):

- асинхронные проверки кода Python (команды `pylint`, `pyflakes`, `pep8`, `mccabe`) в любой комбинации;

- рефакторинг кода и автозаполнение с помощью библиотеки rope;
- быстрое свертывание кода Python;
- поддержка инструмента virtualenv;
- возможность выполнять поиск в документации Python и запускать код Python;
- автоматическое исправление ошибок pep8.

Emacs

Emacs — еще один мощный текстовый редактор. Теперь он имеет графический интерфейс, но его все еще можно запустить в консоли. Его можно настраивать с помощью Lisp. Затратив немного усилий, вы можете превратить этот редактор в IDE для Python. Им пользуются мазохисты и Реймонд Хеттингер (Raymond Hettinger)[1] (http://pyvideo.org/speaker/138/raymond-hettinger).

Emacs написан на Lisp, выпущен в 1976 году Ричардом Столлманом (Richard Stallman) и Гаем Ли Стилом (Guy L. Steele). Встроенная функциональность включает в себя удаленное редактирование (с помощью FTP), календарь, возможность отправлять/читать почту и даже сжатие (нажмите клавишу Esc, затем введите x и `doctor`). Среди популярных надстроек упомянем плагин YASnippet, необходимый для соотнесения пользовательских сниппетов кода и горячих клавиш, и Tramp, предназначенный для отладки. Его можно расширять с помощью собственного диалекта языка Lisp, elisp plus.

Если вы уже используете Emacs, в статье *Python Programming in Emacs*, размещенной в EmacsWiki по адресу http://emacswiki.org/emacs/PythonProgrammingInEmacs, прочитайте советы о том, какие пакеты и конфигурация Python понадобятся. Те, кто еще не работал с Emacs, могут начать свое знакомство с официального руководства для Emacs (http://bit.ly/gnu-emacs-tutorial).

В данный момент для Emacs существует три основные модификации для работы с Python:

- *python.el* от Фабиана Эзекуеля Галлины (Fabián Ezequiel Gallina), теперь поставляется с Emacs (версия 24.3+), реализует подсветку синтаксиса, отступы, перемещение, взаимодействие с оболочкой и многие другие распространенные особенности Emacs (https://github.com/fgallina/python.el#introduction);
- *Elpy* (http://elpy.readthedocs.org/) от Йоргена Шефера (Jorgen Schäfer) предоставляет полноценную интерактивную среду разработки на базе Emacs, которая включает в себя инструменты для отладки, проверки кода и автозаполнение кода;
- *комплект файлов исходного кода для Python* (https://www.python.org/downloads/source/) поставляется с альтернативной версией, которая располагается в каталоге

[1] Нам нравится Реймонд Хеттингер. Если бы все писали код так, как он рекомендует, мир был бы гораздо лучше.

`Misc/python-mode.el`. Вы можете загрузить его из Интернета как отдельный файл из приложения launchpad (https://launchpad.net/python-mode). Содержит инструменты для распознавания голоса, возможность настройки дополнительных горячих клавиш, а также позволяет создать полноценную IDE для Python (http://www.emacswiki.org/emacs/ProgrammingWithPythonModeDotEl).

TextMate

TextMate (http://macromates.com/) — текстовый редактор с графическим интерфейсом. Создан на базе Emacs и работает только для OS X. Имеет удобный пользовательский интерфейс, найти все его команды не составит труда.

TextMate написан на C++ и был впервые выпущен в 2004 году Алланом Одгардом (Allan Oddgard) и Циараном Уолшем (Ciarбn Walsh). Sublime Text может импортировать сниппеты непосредственно в TextMate, а Code от компании Microsoft — подсветку синтаксиса TextMate.

Сниппеты, написанные на любом языке, можно собирать в связанные группы. Редактор можно расширить с помощью сценариев оболочки: пользователь выделяет текст и отправляет его в качестве входного параметра для сценария нажатием сочетания клавиш Cmd+| («пайп»). Результат работы сценария заменит выделенный текст.

Редактор имеет встроенную подсветку синтаксиса для Swift и Objective C, а также (благодаря Xcode bundle) интерфейс для xcodebuild. Опытный пользователь TextMate не испытает затруднений при написании кода Python. Новым пользователям, не имеющим достаточного опыта написания кода для продуктов компании Apple, стоит начать работу с более новыми кросс-платформенными редакторами, заимствующими многие особенности TextMate.

Atom

Atom (https://atom.io/), по мнению его создателей, — это «уязвимый для хакерских атак текстовый редактор XXI века». Выпущен в 2014 году, написан на CoffeeScript (JavaScript) и Less (CSS), основан на Electron (ранее известном как Atom Shell)[1], который является оболочкой приложения для веб-сервиса GitHub, основанного на io.js и Chromium.

Atom можно расширить с помощью JavaScript и CSS, пользователи могут добавлять сниппеты, написанные на любом языке (включая определения сниппетов в стиле TextMate). Редактор хорошо взаимодействует с GitHub. Он поставляется со встроенной функциональностью по управлению пакетами и множеством пакетов (2000+). Для разработки на Python рекомендуется использовать Linter

[1] Electron — платформа для создания кросс-платформенных настольных приложений с помощью HTML, CSS и JavaScript.

(https://github.com/AtomLinter/Linter) вместе с linter-flake8 (https://github.com/AtomLinter/linter-flake8). Веб-разработчикам также может понравиться Atom development server (https://atom.io/packages/atom-development-server), который запускает небольшой HTTP-сервер и способен показать получившуюся в процессе работы HTML-страницу внутри Atom.

Code

Компания Microsoft анонсировала Code в 2015 году. Это бесплатный текстовый редактор с закрытым исходным кодом в семействе Visual Studio, основанный на Electron, который можно найти на GitHub. Редактор кросс-платформенный, имеет такие же привязки клавиш, что и TextMate.

Code поставляется в качестве расширения API (https://code.visualstudio.com/Docs/extensions/overview) (обратите внимание на VS Code Extension Marketplace, где можно найти существующие расширения (https://code.visualstudio.com/docs/editor/extension-gallery), — он объединяет лучшие, по мнению разработчиков, части TextMate и Atom). Редактор имеет функцию IntelliSense (автозаполнение кода), как и VisualStudio, и поддерживает .Net, C# и F#.

Visual Studio (IDE, родственная текстовому редактору Code) все еще работает только для Windows, даже несмотря на то, что редактор Code кросс-платформенный.

IDE

Многие разработчики используют и текстовые редакторы, и IDE, переключаясь на IDE для работы над более сложными, крупными или совместными проектами. В табл. 3.2 перечислены основные особенности популярных IDE, а в следующих разделах приведена более подробная информация о каждом из них.

Одна из особенностей, которую часто называют веской причиной пользоваться только IDE (помимо автозаполнения кода и инструментов для отладки), — возможность быстро переключаться между интерпретаторами Python (например, с Python 2 на Python 3 или IronPython), она доступна в бесплатных версиях всех IDE, перечисленных в табл. 3.2. Visual Studio предлагает эту функциональность на всех уровнях[1].

Дополнительная функциональность — это инструменты для работы с системами тикетов, инструменты развертывания (например, Heroku или Google App Engine), инструменты для взаимодействия и прочие функции, которые можно использовать во фреймворках, связанных с веб-разработкой, например Django.

[1] https://github.com/Microso/PTVS/wiki/Features-Matrix.

Таблица 3.2. Первый взгляд на IDE

Инструмент	Доступность	Причина использовать
PyCharm/Intellij IDEA	• Открытый API/платная версия для профессионалов. • Открытый исходный код/бесплатная версия для сообщества. • OS X, Linux, Windows	• Практически идеальное автозаполнение кода. • Хорошая поддержка виртуальных сред. • Хорошая поддержка веб-фреймворков (в платной версии)
Aptana Studio 3 / Eclipse + LiClipse + PyDev	• Открытый исходный код/бесплатное ПО. • OS X, Linux, Windows	• Вам нравится Eclipse. • Поддержка Java (LiClipse/Eclipse)
WingIDE	• Открытый исходный код/бесплатный пробный период. • OS X, Linux, Windows	• Отличный отладчик (для веб-приложений) — лучший среди перечисленных здесь IDE. • Можно расширять с помощью Python
Spyder	• Открытый исходный код/бесплатное ПО. • OS X, Linux, Windows	• Анализ данных: интегрирован IPython вместе с NumPy, SciPy и matplotlib. • IDE по умолчанию в популярных научных дистрибутивах Python: Anaconda, Python(x,y) и WinPython
NINJA-IDE	• Открытый исходный код/можно вносить пожертвования. • OS X, Linux, Windows	• Имеет небольшой размер. • Сконцентрирован на Python
Komodo IDE	• Открытый API/текстовый редактор (Komodo Edit) имеет открытый исходный код. • OS X, Linux, Windows	• Python, PHP, Perl, Ruby, Node. • Расширения основаны на дополнениях для Mozilla
Eric (the Eric Python IDE)	• Открытый исходный код/можно вносить пожертвования. • OS X, Linux, Windows	• Ruby + Python. • Небольшой по размеру. • Отличный отладчик (научный) — можно выполнять отладку одного потока и выполнять другие
Visual Studio (Community)	• Открытый API/бесплатная версия для сообщества. • Платная версия для профессионалов и предприятий. • Только для Windows	• Отличная интеграция с инструментами и языками компании Microsoft. • Фантастическое автозаполнение кода посредством IntelliSense.

Инструмент	Доступность	Причина использовать
		• Управление проектами и поддержка при развертывании, включая инструменты для планирования спринтов и шаблоны манифестов в версии Enterprise. Примечание. Имеется недостаток: нельзя использовать виртуальные среды во всех версиях, кроме Enterprise (самой дорогой)

PyCharm/IntelliJ IDEA

PyCharm (http://www.jetbrains.com/pycharm/) — наша любимая IDE для Python. В качестве основных причин использовать именно ее можно привести практически идеальные инструменты автозаполнения кода, а также качество инструментов для веб-разработки. Участники научного сообщества рекомендуют бесплатную версию (которая не имеет инструментов для веб-разработки), поскольку она вполне им подходит, однако чаще они выбирают Spyder.

PyCharm разрабатывается компанией JetBrains, также известной как IntelliJ IDEA. Представляет собой проприетарную IDE для Java, которая конкурирует с Eclipse. PyCharm (выпущена в 2010 году) и IntelliJ IDEA (выпущена в 2001-м) имеют общую базу кода, и большую часть функциональности PyCharm можно использовать в IntelliJ благодаря бесплатной надстройке на Python (http://bit.ly/intellij-python).

JetBrains рекомендует работать с PyCharm, если вам нужен простой пользовательский интерфейс, или с IntelliJ IDEA, если вы хотите изучать функции Jython, выполнять задачи на разных языках или преобразовывать код на Java в код на Python. (PyCharm тоже работает с Jython, но только как возможный вариант интерпретатора.) Эти две IDE имеют разные лицензии, поэтому перед покупкой нужно сделать выбор.

IntelliJ Community Edition и PyCharm Commuity Edition имеют открытый исходный код (лицензия Apache 2.0) и бесплатны.

Aptana Studio 3/Eclipse + LiClipse + PyDev

Eclipse написана на Java, выпущена в 2001 году компанией IBM как открытая и гибкая IDE для Java. PyDev (http://pydev.org/), надстройка Eclipse для разработки на Python, выпущена в 2003-м Алексом Тотиком (Aleks Totic), который впоследствии передал эстафету Фабио Задрожному (Fabio Zadrozny). Это наиболее популярная надстройка Eclipse при разработке на Python.

Несмотря на то что сообщество Eclipse не перечит, когда кто-то голосует за использование IntelliJ IDEA на форумах, где сравниваются эти две IDE, Eclipse все еще считается наиболее распространенной IDE для Java. Это важно для разработчиков на Python, взаимодействующих с инструментами, написанными на Java, поскольку многие популярные инструменты (например, Hadoop, Spark и их проприетарные версии) поставляются с инструкциями и надстройками для разработки с помощью Eclipse.

В Studio 3 от Aptana (http://www.aptana.com/products/studio3.html) встроена версия PyDev, представляющая собой набор надстроек с открытым исходным кодом. Она поставляется с Eclipse (предоставляет IDE для Python (и Django), Ruby (и Rails), HTML, CSS и PHP). Основные направления развития владельца Aptanta, Appcelerator, — Appcelerator Studio, проприетарная мобильная платформа для HTML, CSS и JavaScript, требующая покупки месячной лицензии (как только вы выпустите свое приложение). Они поддерживают и PyDev с Python, но это направление не является приоритетным. Если вам нравится Eclipse и вы в основном работаете с JavaScript, создавая приложения для мобильных платформ и иногда используя Python (особенно если работаете с Appcelerator), Studio 3 от Aptana отлично подойдет.

LiClipse появился на свет благодаря желанию предоставить более качественную поддержку нескольких языков в Eclipse, а также упростить доступ к полностью черным темам (то есть в дополнение к фону для текста меню и границы также будут черными). Этот проприетарный набор надстроек для Eclipse написан Задрожным; часть стоимости лицензии (опционально) идет на то, чтобы PyDev оставался полностью бесплатным и имел открытый исходный код (лицензия EPL, как и у Eclipse). Поставляется вместе с PyDev, поэтому пользователям Python не нужно устанавливать его самостоятельно.

WingIDE

WingIDE (http://wingware.com/) — это IDE для Python. Вполне возможно, что это вторая по популярности IDE для Python после PyCharm. Работает в Linux, Windows и OS X. Инструменты отладки весьма функциональны (среди них есть инструмент для отладки шаблонов Django).

В качестве причин использовать WingIDE называются отладчик и небольшой объем этой IDE.

Среда Wing выпущена в 2000 году компанией Wingware, написана на Python, C и C++. Поддерживает расширения, но пока еще не имеет репозитория надстроек, поэтому ее пользователям приходится искать существующие пакеты в блогах, а также в учетных записях GitHub.

Spyder

Spyder (https://github.com/spyder-ide/spyder) (расшифровывается как Scientific PYthon Development EnviRonment — научная среда разработки для Python) — это IDE, предназначенная для работы с научными библиотеками Python. Написана на Python Карлосом Сирдобой (Carlos Cyrdoba), имеет открытый исходный код (лицензия MIT) и предлагает такие возможности, как автозаполнение кода, подсветка синтаксиса, обозреватель классов и функций, исследование объектов. Доступ к другой функциональности можно получить с помощью надстроек от сообщества.

Spyder можно интегрировать с библиотеками pyflakes, pylint и rope. Поставляется с библиотеками NumPy, SciPy, IPython и Matplotlib, а также с популярными дистрибутивами Python для науки — Anaconda, Python(x, y) и WinPython.

NINJA-IDE

NINJA-IDE (http://www.ninja-ide.org/) (название представляет собой рекурсивный акроним фразы Ninja-IDE Is Not Just Another IDE (Ninja-IDE — это не просто еще одна IDE)) — кросс-платформенная IDE, разработанная для сборки приложений на Python. Работает в Linux/X11, Mac OS X и Windows. Установщики для этих платформ можно загрузить с сайта NINJA-IDE.

NINJA-IDE разработана с использованием Python и Qt, имеет открытый исходный код (лицензия GPLv3) и специально создана легковесной. В версии без надстроек самой популярной особенностью является подсветка проблемного кода при запуске проверки кода или при отладке, а также возможность предпросмотра веб-страниц во встроенном браузере. Вы можете расширить ее возможности с помощью Python, имеется репозиторий надстроек (пользователи могут добавлять только необходимые инструменты).

Разработка на какое-то время замедлилась, но новую версию NINJA-IDE v3 планируется выпустить в 2016 году и в данный момент все еще идут активные переговоры насчет listserv для NINJA-IDE (http://bit.ly/ninja-ide-listserv). Для многих членов сообщества, включая команду разработчиков, родным языком является испанский.

Komodo IDE

Komodo IDE (http://www.activestate.com/komodo-ide) разработана компанией ActiveState, является коммерческой IDE для Windows, Mac и Linux. Текстовый редактор (https://github.com/Komodo/KomodoEdit) имеет альтернативу с открытым исходным кодом (под общественной лицензией Mozilla).

Выпущена в 2000 году компанией ActiveState, в ней используется база кода Mozilla и Scintilla. Ее можно расширить с помощью надстроек для Mozilla. Поддерживает языки Python, Perl, Ruby, PHP, Tcl, SQL, Smarty, CSS, HTML и XML. Komodo Edit не имеет отладчика (однако доступен в виде надстройки). IDE не поддерживает виртуальные среды, но позволяет пользователю выбрать, какой интерпретатор Python использовать. Django поддерживается не так широко, как в WingIDE, PyCharm или Eclipse + PyDev.

Eric (the Eric Python IDE)

Eric — это IDE с открытым исходным кодом (лицензия GPLv3), которая активно разрабатывается уже более десяти лет. Написана на Python на базе набора инструментов Qt GUI, в который интегрирован редактор Scintilla. Названа в честь Эрика Айдла (Eric Idle), члена группы «Монти Пайтон» (Monty Python), а также в знак уважения к IDLE IDE, поставляющейся с дистрибутивами Python.

В числе ее особенностей — автозаполнение кода, подсветка синтаксиса, поддержка системы контроля версий, поддержка Python 3, интегрированный браузер, оболочка Python, встроенный отладчик и гибкая система надстроек. Не имеет отдельных инструментов для работы с веб-фреймворками.

Как NINJA-IDE и Komodo IDE, среда Eric специально создана легковесной. Преданные пользователи верят, что ее отладчик самый лучший, поскольку, помимо всего прочего, он имеет возможность остановить один поток и выполнять отладку для него, не прекращая при этом другие потоки. Если вы хотите задействовать Matplotlib для интерактивного создания графиков в этой IDE, используйте бэкенд Qt4:

```
# Сначала введите эти строки:
import matplotlib
matplotlib.use('Qt4Agg')

# А затем pyplot будет использовать бэкенд Qt4:
import matplotlib.pyplot as plt
```

Ссылка указывает на самую последнюю версию документации для Eric IDE (http://eric-ide.python-projects.org/eric-documentation.html). Практически все пользователи, которые оставляют положительные отзывы на веб-странице этой IDE, состоят в сообществе, занимающемся научными вычислениями (например, погодными моделями или гидродинамическим моделированием).

Visual Studio

Профессиональные программисты, работающие с продуктами компании Microsoft в операционной системе Windows, захотят воспользоваться Visual Studio (https://www.visualstudio.com/products). Она написана на C++ и C#, первая версия выпущена

в 1995 году. В конце 2014 года первая версия Visual Studio Community Edition стала доступна бесплатно для некоммерческих разработчиков.

Если вы планируете работать в основном с корпоративным ПО и использовать такие продукты компании Microsoft, как C# и F#, эта IDE идеальна для вас.

Убедитесь, что устанавливаете Python Tools for Visual Studio (PTVS) (https://www.visualstudio.com/en-us/features/python-vs.aspx) (этот вариант не задан по умолчанию в списке пользовательских вариантов установки). Инструкции по установке Visual Studio и о том, что делать после нее, можно найти на вики-странице PTVS (https://github.com/Microsoft/PTVS/wiki/PTVS-Installation).

Улучшенные интерактивные инструменты

Инструменты, перечисленные в этом разделе, повышают возможности интерактивной оболочки. IDLE, по сути, является IDE, но она не была включена в предыдущий раздел, поскольку многие не считают ее достаточно надежной для того, чтобы использовать для производственных проектов, как другие IDE. Однако она отлично подходит для обучения. IPython по умолчанию встроен в Spyder (может быть встроен и в другие IDE). Они не заменяют интерпретатор Python, скорее подавляют выбранную пользователем оболочку интерпретатора с помощью дополнительных инструментов и функциональности.

IDLE

IDLE (Integrated Development and Learning Environment — интегрированная среда для разработки и обучения; идет также отсылка к фамилии Эрика Айдла (Eric Idle)) (http://docs.python.org/library/idle.html#idle) — часть стандартной библиотеки Python. Поставляется вместе с Python.

IDLE полностью написана на Python Гвидо ван Россумом (Guido van Rossum) (BDFL для Python — Benevolent Dictator for Life (великодушный пожизненный диктатор)), использует набор инструментов Tkinter GUI. Хотя IDLE не подходит для полноценной разработки на базе Python, в ней полезно пробовать запускать небольшие сниппеты кода и экспериментировать с различными особенностями языка.

Предоставляет следующие возможности:

- окно оболочки Python (интерпретатора);
- многооконный текстовый редактор, который выделяет код Python разными цветами;
- минимальные возможности отладки.

IPython

IPython (http://ipython.org/) — полезный набор инструментов, который поможет вам максимально задействовать интерактивную часть Python.

Перечислим его основные компоненты:

- мощные оболочки Python (на базе консоли и Qt);
- блокнот на базе Интернета, имеющий такую же основную функциональность, как и терминальная оболочка, а также поддерживающий мультимедиа, текст, код, математические выражения и встроенные графики;
- поддержка интерактивной визуализации данных (при соответствующем конфигурировании ваши графики, созданные в Matplotlib, будут появляться в окнах) и применения инструментов графического пользовательского интерфейса;
- гибкие встраиваемые интерпретаторы, предназначенные для загрузки ваших проектов;
- инструменты для выполнения высокоуровневых и интерактивных параллельных вычислений.

Для установки IPython введите следующий код в терминальной оболочке или в PowerShell:

```
$ pip install ipython
```

bpython

bpython (http://bpython-interpreter.org/) — альтернативный интерфейс для интерпретатора Python, который работает в Unix-подобных системах. Для него характерны:

- встроенная подсветка синтаксиса;
- автоматическое создание отступов и автозаполнение кода;
- ожидаемый список параметров для любой функции Python;
- функция «перемотки», которая позволяет выделить последнюю строку кода из памяти и повторно оценить ее;
- возможность отправить введенный код в pastebin (чтобы поделиться им в сети);
- возможность сохранить введенный код в файл.

Для установки bpython введите в терминальной оболочке следующий код:

```
$ pip install bpython
```

Инструменты изоляции

В этом разделе приводится подробная информация о наиболее популярных инструментах изоляции — от virtualenv, который изолирует среды Python друг от друга, до Docker, который позволяет создать целую виртуальную систему.

Эти инструменты предоставляют разные уровни изоляции между запущенным приложением и средой его хоста, позволяют протестировать и отладить код для разных версий Python и зависимых библиотек, могут использоваться в качестве устойчивой среды развертывания.

Виртуальные среды

Виртуальные среды Python отдельно хранят требующиеся для разных проектов зависимости. При установке нескольких сред Python ваш глобальный каталог `site-packages` (место, где хранятся установленные пользователем пакеты Python) останется упорядоченным и вы сможете работать над проектом, который требует наличия фреймворка Django 1.3, в то же время поддерживая проект, требующий Django 1.0.

Команда `virtualenv` позволяет достичь этого, создавая отдельный каталог, содержащий гибкую ссылку на исполняемый файл Python, копию pip и место для библиотек Python (добавляет его в начало переменной среды `PATH` при активизации, а затем возвращает переменную среды в исходное состояние при деактивизации). Вы также можете использовать установленные вместе с системой версию Python и библиотеки, указав требуемые параметры командной строки.

Вы не можете переместить виртуальную среду после ее создания — пути в исполняемых файлах четко указывают на текущий абсолютный путь к интерпретатору, который располагается в каталоге виртуальной среды bin/.

Создание и активизация виртуальной среды

Установка и активизация виртуальных сред Python отличается в разных операционных системах.

Mac OS X и Linux. Вы можете указать версию Python с помощью аргумента `--python`. Далее используйте сценарий активизации, чтобы установить значение переменной среды `PATH` при входе в виртуальную среду:

```
$ cd мой_каталог_проекта
$ virtualenv --python python3 my-venv
$ source my-venv/bin/activate
```

Windows. Нужно настроить политику выполнения для системы (если вы еще этого не сделали), чтобы разрешить запуск сценариев, созданных локально[1].

Запустите PowerShell от имени администратора и введите следующий код:

```
PS C:\> Set-ExecutionPolicy RemoteSigned
```

[1] Если вам больше нравится, используйте команду Set-ExecutionPolicy AllSigned.

Ответьте «Y» на появившийся вопрос и выйдите, после чего в обычной версии PowerShell создайте виртуальную среду:

```
PS C:\> cd мой_каталог_проекта
PS C:\> virtualenv --python python3 my-venv
PS C:\> .\my-venv\Scripts\activate
```

Добавление библиотек в виртуальную среду

Как только вы активизировали виртуальную среду, первым исполняемым файлом pip будет тот, который расположен в только что созданном каталоге `my-venv`. Этот файл установит библиотеки в следующую папку:

- `my-venv/lib/python3.4/site-packages/` (в системах POSIX[1]);
- `my-venv\Lib\site-packages` (в Windows).

При сборке собственных пакетов или проектов для заказчиков можете использовать следующую команду, когда виртуальная среда активна:

```
$ pip freeze > requirements.txt
```

Она позволяет записать все текущие установленные пакеты (которые, как мы надеемся, также являются зависимостями проекта) в файл с именем `requirements.txt`. Взаимодействующие участники могут установить все зависимости в свою собственную виртуальную среду при наличии файла `requirements.txt`, введя следующую команду:

```
$ pip install -r requirements.txt
```

Команда `pip` установит перечисленные зависимости, переопределяя зависимости, указанные в подпакетах, в том случае, если возникли конфликты. Зависимости, указанные в файле `requirements.txt`, предназначены для установки всей среды Python. Для того чтобы установить зависимости при распространении библиотеки, для функции `setup()`, размещенной в файле `setup.py`, лучше всего использовать аргумент с ключевым словом `install_requires`.

Тщательно следите за тем, чтобы не вызвать команду pip install -r requirements.txt за пределами виртуальной среды. Если версия какой-нибудь библиотеки, указанной в файле requirements.txt, будет отличаться от той, что установлена на вашем компьютере, pip изменит ее на ту, которая указана в файле requirements.txt.

[1] POSIX расшифровывается как Portable Operating System Interface (портируемый интерфейс операционных систем). Включает набор стандартов IEEE, указывающих ОС ее поведение и интерфейс для простых команд оболочки, ввода/вывода, работы с потоками и для других сервисов и утилит. Большая часть дистрибутивов Linux и Unix совместима с POSIX, Darwin (операционная система, лежащая в основе Mac OS X и iOS), была совместима с версии Leopard (10.5). Фраза «система POSIX» означает систему, совместимую с POSIX.

Деактивизация виртуальной среды

Чтобы вернуться к обычным системным настройкам, введите следующую команду:

```
$ deactivate
```

Для получения более подробной информации смотрите документацию для виртуальных сред (http://bit.ly/virtualenv-guide), официальную документацию для virtualenv (https://virtualenv.pypa.io/en/latest/userguide.html) или официальное руководство по упаковке для Python (https://packaging.python.org/). Пакет pyvenv, который распространяется как часть стандартной библиотеки Python в версиях 3.3 и выше, не заменяет virtualenv (фактически является зависимостью для virtualenv), поэтому эти инструкции работают для всех версий Python.

pyenv

Инструмент pyenv (https://github.com/yyuu/pyenv) позволяет работать с несколькими версиями интерпретаторов Python одновременно. Это решает проблему, возникающую при наличии нескольких проектов, когда каждый требует разных версий Python, но вам все еще придется использовать виртуальные среды в том случае, если в библиотеках возникнет конфликт зависимостей (например, потребуются разные версии Django). Вы можете установить Python 2.7 для совместимости с одним из проектов и при этом применять в качестве интерпретатора по умолчанию Python 3.5. Инструмент pyenv не ограничен только версиями CPython, он также установит интерпретаторы PyPy, Anaconda, Miniconda, Stackless, Jython и IronPython.

Работа pyenv заключается в том, что он заполняет каталог `shims` вспомогательной версией интерпретатора Python и исполняемыми файлами вроде `pip` и `2to3`. Эти файлы можно найти, если каталог находится в начале переменной среды `$PATH`. Вспомогательная функция — это проходная функция, которая интерпретирует текущую ситуацию и выбирает самую подходящую функцию для выполнения желаемой задачи. Например, когда система ищет программу с именем python, она сначала заглядывает внутрь каталога `shims` и использует вспомогательную версию, которая в свою очередь передает команду `pyenv`. После этого `pyenv` определяет, какая версия Python должна быть запущена, основываясь на переменных среды, файлах версии с расширением `*.python` и глобальных значениях по умолчанию.

Для виртуальных сред применяется надстройка pyenv-virtualenv (https://github.com/yyuu/pyenv-virtualenv): автоматизирует создание различных сред, а также позволяет использовать существующие инструменты pyenv для переключения между ними.

Autoenv

Autoenv (https://github.com/kennethreitz/autoenv) позволяет легко управлять различными настройками среды за пределами области видимости virtualenv. Переопределяет команду оболочки `cd` таким образом, что, когда вы переходите в каталог, содержащий

файл с расширением `.env` (например, устанавливая значение переменной среды `PATH` с помощью URL для базы данных), Autoenv автомагически активизирует среду. Когда вы выходите из каталога, вызвав эту же команду, все отменяется (не работает в Windows PowerShell).

Вы можете установить autoenv в Mac OS X с помощью команды `brew`:

```
$ brew install autoenv
```

Или в Linux:

```
$ git clone git://github.com/kennethreitz/autoenv.git ~/.autoenv
$ echo 'source ~/.autoenv/activate.sh' >> ~/.bashrc
```

А затем открыть новое окно консоли.

virtualenvwrapper

Инструмент virtualenvwrapper (http://bit.ly/virtualenvwrapper-docs) предлагает набор команд, который расширяет виртуальные среды Python, чтобы ими было легче управлять. Он помещает все ваши виртуальные среды в один каталог и предоставляет пустые функции перехвата (их можно запустить до или после создания/активизации виртуальной среды или проекта, например функция перехвата может установить переменные среды путем поиска файла с расширением `.env` внутри каталога).

Проблема с размещением функций с установленными объектами заключается в том, что пользователь должен каким-то образом получить доступ к данным сценариям, чтобы полностью скопировать среду на другую машину. Это может пригодиться для общего сервера, если все среды помещены в единый каталог и доступны нескольким пользователям.

Для того чтобы пропустить полные инструкции по установке virtualenvwrapper (http://bit.ly/virtualenvwrapper-install), сначала убедитесь, что у вас уже установлен virtualenv. Затем в OS X или Linux введите следующую строку в командную консоль:

```
$ pip install virtualenvwrapper
```

Используйте команду `pip install virtualenvwrapper`, если работаете с Python 2, добавьте эту строку в ваш профиль:

```
export VIRTUALENVWRAPPER_PYTHON=/usr/local/bin/python3
```

Далее введите следующую строку в ваш профиль `~/.bash_profile` или любой другой профиль оболочки:

```
source /usr/local/bin/virtualenvwrapper.sh
```

Наконец, закройте текущее окно консоли и откройте новое, чтобы активизировать ваш новый профиль. Теперь virtualenvwrapper доступен.

В Windows следует использовать virtualenvwrapper-win. После установки virtualenv введите следующий код:

```
PS C:\> pip install virtualenvwrapper-win
```

На обеих платформах наиболее часто используются следующие команды:

- `mkvirtualenv` *`my_venv`* — создает виртуальную среду в каталоге `~/.virtualenvs/` *`my_venv`*. В Windows среда *`my_venv`* будет создана в каталоге, который можно определить, введя команду `%USERPROFILE%\Envs` в командной строке. Это местоположение можно изменить с помощью переменной среды `$WORKON_HOME`;
- `workon` *`my_venv`* — активизирует виртуальную среду или переключает вас на указанную среду;
- `deactivate` — деактивизирует виртуальную среду;
- `rmvirtualenv` *`my_venv`* — удаляет виртуальную среду.

Инструмент virtualenvwrapper предоставляет возможность заполнения имен сред путем нажатия клавиши Tab (может пригодиться, если у вас множество сред и трудно запомнить их имена). Немало других полезных функций задокументировано в полном списке команд virtualenvwrapper (http://bit.ly/virtualenvwrapper-command).

Buildout

Buildout (http://www.buildout.org/en/latest/) — это фреймворк для Python, который дает возможность создавать *рецепты*. Это модули Python, содержащие произвольный код (обычно системные вызовы для создания каталогов или код, позволяющий проверить и построить исходный код либо добавить в проект элементы, написанные не на Python, например базу данных или сервер). Установите его с помощью команды `pip`:

```
$ pip install zc.buildout
```

Проекты, использующие Buildout, будут содержать `zc.buildout` и необходимые им рецепты в файле `requirements.txt` (либо включат пользовательские рецепты в исходный код), а также конфигурационный файл `buildout.cfg` и сценарий `bootstrap.py` в каталоге верхнего уровня. Если вы запустите сценарий, введя команду `python bootstrap.py`, он прочтет конфигурационный файл, чтобы определить, какие рецепты нужно использовать, а также настройки конфигурации для каждого рецепта (например, определенные флаги компилятора и флаги для связывания библиотек).

Buildout позволяет портировать проекты Python, включающие фрагменты, написанные не на Python (другой пользователь может воссоздать такую же среду). В этом отличие от сценариев-перехватчиков в virtualenvwrapper, которые нужно скопировать и передать вместе с файлом `requirements.txt`, чтобы можно было воссоздать виртуальную среду. Содержит все необходимое для установки архивов

egg[1], что можно пропустить в новых версиях Python, которые используют архивы wheels. Обратитесь к руководству Buildout (http://www.buildout.org/en/latest/docs/tutorial.html) для получения более подробной информации.

Conda

Инструмент Conda (http://conda.pydata.org/docs/) похож на pip, virtualenv и Buildout одновременно. Поставляется с дистрибутивом Anaconda и является его менеджером пакетов по умолчанию. Его можно установить с помощью `pip`:

```
$ pip install conda
```

А pip — с помощью `conda`:

```
$ conda install pip
```

Пакеты хранятся в разных репозиториях (`pip` получает их из http://pypi.python.org, а `conda` из https://repo.continuum.io/), имеют разные форматы, поэтому эти инструменты невзаимозаменяемы.

В таблице по адресу http://bit.ly/conda-pip-virtualenvl, созданной компанией Continuum (создателями Anaconda), приводится сравнение трех доступных вариантов: conda, pip и virtualenv.

Инструмент conda-build, аналог Buildout от компании Continuum, может быть установлен на всех платформах, если ввести следующее:

```
conda install conda-build
```

Как и в Buildout, формат файла конфигурации conda-build называется рецептом (не ограничен использованием только лишь инструментов Python). В отличие от Buildout, код указан в сценарии оболочки (это не код Python). Конфигурация приводится в формате YAML[2] (это язык разметки, который понимают и люди, и машины), а не в формате ConfigParser (https://docs.python.org/3/library/configparser.html).

---

[1] Egg — это ZIP-архив с особой структурой. Эти архивы были заменены на архивы wheels в PEP 427 (https://www.python.org/dev/peps/pep-0427/). Были представлены популярной библиотекой упаковки Setuptools (сейчас библиотека по умолчанию), предлагающей полезный интерфейс для distutils (https://docs.python.org/3/library/distutils.html) из стандартной библиотеки Python. Вы можете прочесть об особенностях этих форматов в разделе Wheel vs Egg (https://packaging.python.org/en/latest/wheel_egg/) руководства Python Packaging User.

[2] YAML (https://en.wikipedia.org/wiki/YAML) расшифровывается как YAML Ain't Markup Language — «не просто еще один язык разметки».

Основное преимущество conda перед pip и virtualenv оценят пользователи Windows — библиотеки Python, созданные как расширения на C, могут быть представлены в формате wheels (или в другом), но они практически всегда присутствуют в каталоге пакетов Anaconda (http://docs.continuum.io/anaconda/pkg-docs). Если пакет недоступен через conda, можно установить pip, а затем — пакеты, которые размещаются в PyPI.

Docker

Инструмент Docker (https://www.docker.com/) помогает изолировать среду (как virtualenv, conda или Buildout), но вместо того, чтобы предоставлять виртуальную среду, предлагает *контейнер Docker*. Контейнеры проще изолировать, чем среды. Например, вы можете запустить несколько контейнеров — и у каждого будет свой сетевой интерфейс, правила брандмауэра и имя хоста. Эти контейнеры управляются отдельной утилитой Docker Engine (https://docs.docker.com/engine/), которая координирует доступ к лежащим в их основе операционным системам. Если вы запускаете контейнеры Docker в OS X, Windows или на удаленном хосте, понадобится Docker Machine (https://docs.docker.com/machine/) (позволяетт взаимодействовать с виртуальными машинами[1], запущенными в Docker Engine).

Контейнеры Docker изначально были основаны на контейнерах Linux Containers, которые были связаны с командой оболочки `chroot` (https://en.wikipedia.org/wiki/Chroot).

`chroot` — это подобие команды `virtualenv` на системном уровне: позволяет сделать так, чтобы корневой каталог (/) располагался по адресу, указанному пользователем, а не в реальном корневом каталоге (это предоставляет пользователю отдельное пространство (https://en.wikipedia.org/wiki/User-space)).

Docker больше не использует `chroot` и даже Linux Containers (позволяет включить в число доступных образов Docker машины Citrix и Solaris), но принцип работы контейнеров Docker Containers не изменился. Их конфигурационные файлы называются Dockerfiles (https://docs.docker.com/engine/reference/builder/), с их помощью создаются образы Docker (https://docs.docker.com/engine/userguide/containers/dockerimages/), которые можно разместить в Docker Hub (https://docs.docker.com/docker-hub/), репозитории пакетов Docker (аналогичен PyPI).

Корректно сконфигурированные образы Docker занимают гораздо меньше места, чем среды, созданные с помощью Buildout или conda, поскольку Docker использует файловую систему AUFS, хранящую «разность» образа, а не сам образ. Поэтому,

[1] Виртуальная машина — это приложение, которое эмулирует компьютерную систему путем имитации желаемого аппаратного обеспечения и предоставления требуемой операционной системы для компьютера-хоста.

если вы хотите построить и протестировать свой пакет для нескольких версий зависимости, можете создать основной образ Docker с виртуальной средой[1] (средой Buildout или conda), содержащей все остальные зависимости.

Вы унаследуете от этого образа все остальные образы, добавив на последнем уровне одну изменяющуюся зависимость. В результате все унаследованные контейнеры будут содержать только отличающиеся библиотеки, разделяя при этом содержимое основного образа. Для получения более подробной информации обратитесь к документации Docker по адресу https://docs.docker.com/.

[1] Виртуальная среда внутри контейнера Docker изолирует вашу среду Python, сохраняя версию Python операционной системы, которая может понадобиться для ваших приложений. Поэтому прислушайтесь к нашему совету ничего не устанавливать с помощью pip (или чего-то еще) в системный каталог Python.

Часть II. Переходим к делу

Теперь у нас есть интерпретатор Python, виртуальные среды и редактор или IDE, так что мы готовы заняться делом. В этой части книги мы не будем изучать язык (в разделе «Изучаем Python» приложения перечислены отличные ресурсы, которые помогут вам в этом). Мы хотим, чтобы после прочтения вы почувствовали себя настоящим программистом Python, знающим все хитрости лучших питонистов. В эту часть входят следующие главы.

- Глава 4 «Пишем отличный код». Мы кратко рассмотрим стиль, соглашения, идиомы и подводные камни.
- Глава 5 «Читаем отличный код». Мы проведем для вас экскурсию по нашим любимым библиотекам (может, это вдохновит вас на дальнейшее чтение хорошего кода?).
- Глава 6 «Отправляем отличный код». Мы кратко поговорим о Python Packaging Authority и о том, как загружать бинарные файлы в PyPI, рассмотрим варианты сборки и отправки исполняемых файлов.

4 Пишем отличный код

В этой главе продемонстрированы лучшие приемы написания отличного кода Python. Мы рассмотрим соглашения, связанные со стилем написания кода, а также правила хорошего тона, связанные с журналированием, перечислим основные отличия между доступными лицензиями для открытого исходного кода. Это поможет вам писать код, который впоследствии можно будет легко использовать и расширять.

Стиль кода

Питонисты (ветераны разработки на Python) рады тому, что их язык настолько понятен, — люди, которые никогда не занимались разработкой, способны разобраться в работе программы при чтении ее исходного кода. Легкость чтения лежит в основе дизайна Python (важно понимать, что написанный код будет прочитан много раз).

Одна из причин, почему код Python прост для понимания, заключается в информативном руководстве по стилю написания кода (оно представлено в двух Предложениях по развитию Python (Python Enhancement Proposal) PEP 20 и PEP 8; о них скажем пару слов) и питонских идиомах. Если питонист указывает на фрагмент кода и говорит, что он не питонский, это обычно означает, что строки не соответствуют распространенным принципам и не являются читаемыми. Конечно, «глупая последовательность — пугало маленьких умов»[1]. Педантичное следование PEP может снизить читаемость и понятность.

---

[1] Цитата изначально приведена Ральфом Уолдо Эмерсоном (Ralph Waldo Emerson) в эссе Self-Reliance. Присутствует в PEP 8 для того, чтобы подтвердить, что здравый смысл важнее руководства по стилю. Например, гармоничный код и существующие соглашения перевесят строгое следование PEP 8.

PEP 8

PEP 8 де-факто представляет собой руководство по стилю написания кода Python. В нем рассматриваются соглашения по именованию, структура кода, пустые области (табуляция против пробелов) и другие аналогичные темы.

Мы рекомендуем изучить его. Все сообщество Python старается следовать принципам, изложенным в этом документе. Некоторые проекты время от времени могут отступать от него, а другие (вроде Requests — http://bit.ly/reitz-code-style) — добавлять поправки к рекомендациям.

Писать код с учетом принципов PEP 8 — хорошая идея (помогает разработчикам создавать более стабильный код). С помощью программы pep8 (https://github.com/jcrocholl/pep8), которая запускается из командной строки, можно проверить код на соответствие принципам PEP 8. Для установки этой программы введите в терминале такую команду:

```
$ pip3 install pep8
```

Рассмотрим пример того, что вы можете увидеть при запуске команды `pep8`:

```
$ pep8 optparse.py
optparse.py:69:11: E401 multiple imports on one line
optparse.py:77:1: E302 expected 2 blank lines, found 1
optparse.py:88:5: E301 expected 1 blank line, found 0
optparse.py:222:34: W602 deprecated form of raising exception
optparse.py:347:31: E211 whitespace before '('
optparse.py:357:17: E201 whitespace after '{'
optparse.py:472:29: E221 multiple spaces before operator
optparse.py:544:21: W601 .has_key() is deprecated, use 'in'
```

Большинство недостатков можно легко исправить, рекомендации по их устранению даются в PEP 8. В руководстве по стилю написания кода для Requests приведены примеры хорошего и плохого кода (лишь немного отличаются от оригинального PEP 8).

Инструменты контроля качества кода, о которых мы говорили в разделе «Текстовые редакторы» в главе 3, обычно используют программу pep8, поэтому вы также можете установить один из них для проверки кода внутри редактора или IDE. Или же можете выбрать команду `auto pep8`, которая автоматически переформатирует код согласно PEP 8. Установить ее можно так:

```
$ pip3 install autopep8
```

Чтобы переформатировать файл (перезаписав оригинал), введите следующую команду:

```
$ autopep8 --in-place optparse.py
```

Если вы не добавите флаг `--in-place`, это заставит программу вывести модифицированный код в консоль (или записать в другой файл). Флаг `--aggressive` выполнит

более существенные изменения, его можно применить несколько раз для получения значительного эффекта.

PEP 20 (также известный как «Дзен Питона»)

PEP 20 (https://www.python.org/dev/peps/pep-0020/) (набор принципов для принятия решений в Python) всегда доступен по команде `import this` в оболочке Python. Несмотря на название, PEP 20 содержит 19 афоризмов, а не 20 (последний не был записан).

Реальная история «Дзена Питона» увековечена в статье Барри Уорсоу (Barry Warsaw) *Import this and the Zen of Python* (http://bit.ly/import-this-zen-python).

Дзен Питона. Автор Тим Питерс

Красивое лучше, чем уродливое.

Явное лучше, чем неявное.

Простое лучше, чем сложное.

Сложное лучше, чем запутанное.

Одноуровневое лучше, чем вложенное.

Разреженное лучше, чем плотное.

Читаемость имеет значение.

Особые случаи не настолько особые, чтобы нарушать правила.

При этом практичность важнее безупречности.

Ошибки никогда не должны замалчиваться.

Если не замалчиваются явно.

Встретив двусмысленность, отбрось искушение угадать.

Должен существовать один — и желательно только один — очевидный способ сделать это.

Хотя он поначалу может быть и не очевиден, если вы не голландец.

Сейчас лучше, чем никогда.

Хотя никогда зачастую лучше, чем прямо сейчас.

Если реализацию сложно объяснить — идея плоха.

Если реализацию легко объяснить — идея, возможно, хороша.

Пространства имен — отличная штука! Будем делать их побольше!

Для того чтобы увидеть пример использования каждого из этих афоризмов, обратитесь к презентации Хантера Блэнкса (Hunter Blanks) *PEP 20 (The Zen of Python) by Example* (http://artifex.org/~hblanks/talks/2011/pep20_by_example.pdf). Рэймонд Хеттингер

(Raymond Hettinger) также демонстрирует применение этих принципов в своей речи *Beyond PEP 8: Best Practices for Beautiful, Intelligible Code* (http://bit.ly/beyond-pep-8).

Общие советы

В этом разделе приводятся концепции, связанные со стилем (надеемся, вы с ними согласитесь). Зачастую они применимы и к другим языкам. Некоторые следуют непосредственно из «Дзена Питона», другие основаны на здравом смысле. Они подтверждают наш принцип работы: при написании кода Python выбирать наиболее очевидный способ его представления из имеющихся вариантов.

Явное лучше, чем неявное

В Python предпочтителен наиболее явный способ выражения:

Плохой код	**Хороший код**

`def make_dict(*args):`
`  x, y = args`
`  return dict(**locals())`
	`def make_dict(x, y):`
	`  return {'x': x, 'y': y}`

В примере хорошего кода *x* и *y* явно принимаются от вызывающей стороны, явно возвращается словарь. Возьмите на вооружение полезное правило: другой разработчик должен понять, что делает функция, прочитав ее первую и последнюю строки. В примере плохого кода это правило не выполняется. (Конечно, функцию довольно просто понять, если она состоит всего из двух строк.)

Разреженное лучше, чем плотное

В каждой строке размещайте только одно выражение. Использование сложных выражений (вроде абстракция списков (иначе называют списковыми включениями — list comprehensions)) позволяется и даже поощряется за их краткость и выразительность, но признаком хорошего тона будет размещение отдельных выражений на разных строках. Это поможет создавать более простые для понимания разности[1], когда подобное выражение изменяется:

Плохой код	**Хороший код**
`print('one'); print('two')`	`print('one')` `print('two')`

Продолжение ⇗

[1] Разность — это утилита оболочки, которая сравнивает два файла и показывает отличающиеся строки.

(Продолжение)

Плохой код	Хороший код
`if x == 1: print('one')`	`if x == 1:` `  print('one')`

`if (<complex comparison> and`
`  <other complex comparison>):`
`  # сделать что-нибудь`
	`cond1 = <complex comparison>`
	`cond2 = <other complex comparison>`
	`if cond1 and cond2:`
	`  # сделать что-нибудь`

Повышение читаемости кода среди питонистов ценится гораздо выше, чем увеличение объема на несколько байт (в случае двух-выражений-print-в-одной-строке) или увеличение времени вычисления на несколько миллисекунд (в случае нескольких-условий-в-отдельных-строках). Кроме того, когда группа разработчиков вносит изменения в открытый код, историю изменений хорошего кода проще расшифровать, поскольку изменение в одной строке может воздействовать только на одно выражение.

Ошибки никогда не должны замалчиваться/ Если не замалчиваются явно

Обработка ошибок в Python выполняется с помощью выражения `try`. Пример из пакета HowDoI (более подробно описывается в разделе «HowDoI» в главе 5) Бена Глейтсмана (Ben Gleitzman) показывает, когда замалчивать ошибки приемлемо:

```
def format_output(code, args):
    if not args['color']:
        return code
    lexer = None
    # попробуем отыскать лексеры с помощью тегов Stack Overflow
    # или аргументов query
    for keyword in args['query'].split() + args['tags']:
        try:
            lexer = get_lexer_by_name(keyword)
            break
        except ClassNotFound:
            pass
    # лексер не найден, пробуем угадать
    if not lexer:
        lexer = guess_lexer(code)
    return highlight(code,
                     lexer,
                     TerminalFormatter(bg='dark'))
```

Перед вами часть пакета, который предоставляет сценарий командной строки, позволяющий найти в Интернете (по умолчанию на сайте Stack Overflow) способ выполнить задачу по программированию. Функция `format_output()` подсвечивает синтаксис, просматривая теги вопроса на предмет строки, которую смог разобрать

лексер (также он называется *токенайзером*; теги `python`, `java` или `bash` позволят определить лексер, который нужно использовать для разбиения и подсвечивания кода), а затем, если он даст сбой, пробует определить язык по самому коду. Когда программа достигает оператора `try`, она может пойти по одному из трех путей:

- поток выполнения входит в блок `try` (весь код, расположенный между `try` и `except`), лексер успешно определяется, цикл прерывается, и функция возвращает код, подсвеченный с помощью выбранного лексера;
- лексер не найден, генерируется и обрабатывается исключение `ClassNotFound` — и ничего не происходит. Цикл продолжит выполнение до тех пор, пока не завершится самостоятельно или не будет найден лексер;
- генерируется какое-то другое исключение (например, `KeyboardInterrupt`), которое не обрабатывается и поднимается на верхний уровень, останавливая выполнение.

Часть афоризма «не замалчиваются» препятствует чрезмерному выявлению ошибок. Рассмотрим пример (можете попробовать запустить его *в отдельном окне консоли* — так будет проще прервать выполнение, когда вы во все вникнете):

```
>>> while True:
...     try:
...         print("nyah", end=" ")
...     except:
...         pass
```

Или не пробуйте запускать его. Поскольку для блока `except` не указано конкретное исключение, он будет отлавливать все исключения, в том числе `KeyboardInterrupt` (`Ctrl+C` в консоли POSIX), и игнорировать их. Соответственно, он проигнорирует множество ваших попыток прервать его работу. Это не просто проблема с прерываниями — блок `except` также может скрывать ошибки, что вызовет проблемы в будущем (их станет трудно диагностировать). Поэтому *не замалчивайте ошибки*: всегда явно указывайте имена исключений, которые хотите поймать, и обрабатывайте только их. Если вы хотите просто записать в журнал или как-то еще убедиться в наличии исключения и вызвать его повторно, как в следующем сниппете, тогда все в порядке. Только не замалчивайте ошибки (не обрабатывая их и не вызывая повторно):

```
>>> while True:
...     try:
...         print("ni", end="-")
...     except:
...         print("An exception happened. Raising.")
...         raise
```

Аргументы функций должны быть интуитивно понятными

Ваш выбор при дизайне API определит последующую возможность взаимодействовать с функцией. Аргументы можно передавать в функции четырьмя разными способами.

```
                ❶          ❷          ❸       ❹
def func(positional, keyword=value, *args, **kwargs):
    pass
```

❶ *Позиционные аргументы* обязательны и не имеют значений по умолчанию.

❷ *Аргументы с ключевым словом* необязательны и имеют значения по умолчанию.

❸ *Список с произвольным количеством аргументов* необязателен и не имеет значений по умолчанию.

❹ *Словарь с произвольным количеством аргументов* с ключевым словом необязателен и не имеет значений по умолчанию.

Рассмотрим, когда можно использовать каждый метод передачи аргументов.

- *Позиционные аргументы.* Применяйте этот метод, когда у вас всего несколько аргументов для функции, которые являются частью ее значения и имеют правильный порядок. Например, пользователь без труда вспомнит, что у функций `send(message, recipient)` или `point(x, y)` должны быть два аргумента, а также порядок этих аргументов.

  Антишаблон: при вызове функций можно поменять местами имена аргументов, например так: `send(recipient="World", message="The answer is 42.")` и `point(y=2, x=1)`. Это снижает читаемость. Используйте более понятные вызовы `send("The answer is 42", "World")` и `point(1, 2)`.

- *Аргументы с ключевым словом.* Когда функция имеет более двух или трех позиционных параметров, ее сигнатуру сложнее запомнить. В этом случае можно применить аргументы с ключевым словом, которые имеют значения по умолчанию. Например, более полная версия функции `send` может иметь сигнатуру `send(message, to, cc=None, bcc=None)`. Здесь параметры `cc` и `bcc` являются необязательными и равны `None`, если для них не получено значение.

  Антишаблон: можно отправить аргументы в правильном порядке, но не указывать их имена явно, например `send("42", "Frankie", "Benjy", "Trillian")`, переслав скрытую копию пользователю с именем Триллиан. Можно также передать именованные аргументы в неправильном порядке, например `send("42", "Frankie", bcc="Trillian", cc="Benjy")`. Если у вас нет веской причины делать это, лучше всего использовать вариант, приближенный к определению функции: `send("42", "Frankie", cc="Benjy", bcc="Trillian")`.

Никогда лучше, чем сейчас

Зачастую сложнее удалить опциональный аргумент (и логику внутри функции), который был добавлен на всякий случай и, казалось бы, никогда не используется, чем ввести новый необязательный аргумент и его логику в тот момент, когда они действительно нужны.

- *Список с произвольным количеством аргументов*. Такой список определяется с помощью конструкции `*args`, которая указывает на произвольное количество позиционных аргументов. В теле функции `args` будет играть роль кортежа, состоящего из всех оставшихся позиционных аргументов. Например, функция `send(message, *args)` также может быть вызвана, когда каждый получатель будет представлен отдельным аргументом: `send("42", "Frankie", "Benjy", "Trillian")`. В теле функции конструкция `args` будет равна выражению `("Frankie", "Benjy", "Trillian")`. Хороший пример, иллюстрирующий этот подход, — функция `print`.

  Подводный камень: если функция получает список аргументов одного вида, более понятным будет использование списка или любой другой последовательности. Если функция `send` в этом примере принимает несколько получателей, мы определим ее явно как `send(message, recipients)` и будем вызывать как `send("42", ["Benjy", "Frankie", "Trillian"])`.
- *Словарь с произвольным количеством аргументов с ключевым словом*. Такой словарь определяется с помощью конструкции `**kwargs`, которая указывает на произвольное количество именованных аргументов. В теле функции `kwargs` будет словарем, содержащим все переданные именованные аргументы, которые не были «пойманы» другими аргументами с ключевым словом в сигнатуре функции. Это может быть полезно при журналировании. Средства форматирования на разных уровнях могут принять необходимую им информацию, минуя пользователя.

  Подводный камень: эти мощные приемы нужно применять только в том случае, когда это действительно необходимо. Если же имеется более простая и прозрачная конструкция, то для выражения предназначения функции следует выбрать именно ее.

Имена переменных *args и **kwargs могут (и должны быть) заменены другими, если это более информативно.

Какие аргументы станут позиционными, а какие — необязательными, зависит только от программиста, который пишет функцию. От него также зависит наличие передачи произвольного количества аргументов. В конце концов, должен существовать один (предпочтительно всего один) очевидный способ это сделать. Другие пользователи оценят ваши усилия, если функции, написанные на Python:

- легко прочитать (имя и аргументы не требуют объяснения);
- легко изменить (добавление нового аргумента с ключевым словом не разрушит другие части кода).

Если реализацию сложно объяснить — идея плоха

Python поставляется с богатым набором инструментов (за что его любят хакеры), который позволяет вам делать абсолютно невероятные вещи, например:

- изменять способ создания объектов;
- изменять способ импортирования модулей Python;
- встраивать в Python подпрограммы, написанные на C.

Все эти действия имеют недостатки, поэтому всегда лучше выбирать прямолинейный способ достижения цели. Основной минус: при использовании подобных конструкций снижается читаемость, поэтому то, что вы получаете в результате, должно быть более важным, чем потеря читаемости. Многие инструменты, предназначенные для анализа кода, не смогут работать с таким «волшебным» кодом.

Разработчик Python должен знать о таких практически бесконечных возможностях, поскольку это вселяет уверенность в том, что нерешаемых проблем не существует. Однако важно знать, как и когда применять эти знания *нельзя*.

Как и мастера кун-фу, питонисты знают, как можно убить одним пальцем, и никогда этого не делают.

Мы все — ответственные пользователи

Как уже демонстрировалось, с помощью Python можно делать многое, но некоторые приемы потенциально могут быть опасными. В частности, любой клиентский код может переопределить свойства и методы объекта: в Python нет ключевого слова `private`. Эта философия сильно отличается от той, что присуща высокозащищенным языкам вроде Java, — они имеют множество механизмов, предотвращающих неверное использование. Философия Python сосредоточена во фразе «Мы все — ответственные пользователи».

Это не значит, что ни одно свойство не считается закрытым и что в Python нельзя реализовать инкапсуляцию. Наоборот, вместо того чтобы возводить бетонные стены между своим и чужим кодом, сообщество Python предпочитает полагаться на набор соглашений, которые указывают, к каким элементам нельзя получить доступ напрямую.

Основным соглашением для закрытых свойств и деталей реализации является добавление к именам всех подобных элементов нижнего подчеркивания (например, `sys._getframe`). Если клиентский код нарушает это правило и получает доступ к отмеченным элементам, будет считаться, что любое неверное поведение или проблемы вызваны именно клиентским кодом.

Использование этой концепции всеми одобряется: имя любого метода или свойства, к которым клиентский код не должен получить доступ, должно начинаться с нижнего подчеркивания. Это гарантирует более качественное разделение обязанностей и упрощает внесение изменений в код. Всегда можно сделать закрытое свойство открытым, обратное же действие выполнить гораздо сложнее.

Возвращайте значения из одной точки

Когда сложность функции увеличивается, зачастую вы можете встретить несколько выражений `return` в теле этой функции. Однако для того, чтобы ее было проще понять и прочесть, возвращайте осмысленные значения из минимально возможного количества точек.

Выйти из функции можно в двух случаях: при появлении ошибки или при возвращении значения после того, как функция нормально отработает. Когда функция не может работать корректно, уместно вернуть значение `None` или `False`. В этом случае лучше вернуть значение из функции максимально рано после его обнаружения, дабы упростить структуру функции: весь код, который находится после выражения возврата-в-случае-сбоя, будет считать, что все условия соблюдены, и продолжит вычисление основного результата функции. Необходимы несколько подобных выражений `return`.

Однако везде, где это возможно, имейте только одну точку выхода — сложно выполнять отладку для функций, когда вам сначала нужно определить, какое выражение `return` ответственно за результат. Наличие единой точки выхода из функции также поможет избавиться от некоторых ветвей кода, поскольку наличие пары точек выхода, возможно, намекает на то, что необходимо провести подобный рефакторинг. Код в следующем примере нельзя назвать плохим, но его можно сделать более чистым (как это показано в комментариях):

```
def select_ad(third_party_ads, user_preferences):
    if not third_party_ads:
        return None  # Лучше сгенерировать исключение
    if not user_preferences:
        return None  # Лучше сгенерировать исключение
    # Сложный код, предназначенный для выбора best_ad
    # Из доступных вариантов на основе индивидуальных предпочтений...
    # Постарайтесь устоять перед искушением вернуть best_ad в случае успеха...
    if not best_ad:
        # Запасной план определения best_ad
    return best_ad  # Единая точка выхода, которая поможет обслуживать код
```

Соглашения

Соглашения важны для всех, но это не единственный способ решения задачи. Соглашения, приведенные в этом разделе, довольно распространены, и мы рекомендуем придерживаться их, чтобы сделать свой код более читаемым.

Альтернативы при проверке на равенство

Если вам не нужно явно сравнивать свое значение со значением `True`, `None` или `0`, вы можете добавить его к оператору `if`, как в следующих примерах (см. статью

«Проверка значения на правдивость» (http://docs.python.org/library/stdtypes.html#truth-value-testing) — там представлен список значений, которые расцениваются как `False`).

Плохой код	Хороший код

`if attr == True:`
`  print 'True!'`
	`# Просто проверяем значение`
	`if attr:`
	`  print 'attr is truthy!'`
	`# или проверяем на противоположное значение`
	`if not attr:`
	`  print 'attr is falsey!'`
	`# если вам нужно только значение 'True'`
	`if attr is True:`
	`  print 'attr is True'`

`if attr == None:`
`  print 'attr is None!'`
	`# или явно проверяем на значение None`
	`if attr is None:`
	`  print 'attr is None!'`

Получаем доступ к элементам массива

Используйте синтаксис `x in d` вместо метода `dict.has_key` или передавайте аргумент по умолчанию в метод `dict.get()`.

Плохой код	Хороший код

`>>> d = {'hello': 'world'}`
`>>>`
`>>> if d.has_key('hello'):`
`...  print(d['hello'])`
`# prints 'world'`
`... else:`
`...  print('default_`
`value')`
`...`
`world`
	`>>> d = {'hello': 'world'}`
	`>>>`
	`>>> print d.get('hello', 'default_value')`
	`world`
	`>>> print d.get('howdy', 'default_value')`
	`default_value`
	`>>>`
	`>>> # или:`
	`... if 'hello' in d:`
	`...  print(d['hello'])`
	`...`
	`world`

Манипуляции со списками

Списковые включения — мощный способ работы со списками (для получения более подробной информации обратитесь к соответствующей статье в руководстве *The Python Tutorial* по адресу http://docs.python.org/tutorial/datastructures.html#list-comprehensions). Функции `map()` и `filter()` могут выполнять операции со списками с помощью другого, более выразительного синтаксиса.

Стандартный цикл	Списковое включение

`# Отфильтруем все элементы,`
`# чье значение превышает 4`
`a = [3, 4, 5]`
`b = []`
`for i in a:`
`  if i > 4:`
`  b.append(i)`
	`# Списковое включение выглядит`
	`# прозрачнее`
	`a = [3, 4, 5]`
	`b = [i for i in a if i > 4]`
	`# Или:`
	`b = filter(lambda x: x > 4, a)`

`# Добавим 3 к каждому элементу списка`
`a = [3, 4, 5]`
`for i in range(len(a)):`
`  a[i] += 3`
	`# Здесь также прозрачнее`
	`a = [3, 4, 5]`
	`a = [i + 3 for i in a]`
	`# Или:`
	`a = map(lambda i: i + 3, a)`

Используйте функцию `enumerate()`, чтобы определить свою позицию в списке. Этот вариант выглядит более читаемым, чем создание счетчика, и лучше оптимизирован для итераторов:

```
>>> a = ["icky", "icky", "icky", "p-tang"]
>>> for i, item in enumerate(a):
...     print("{i}: {item}".format(i=i, item=item))
...
0: icky
1: icky
2: icky
3: p-tang
```

Продолжение длинной строки кода

Когда логическая строка кода длиннее принятого значения[1], нужно разбить строку на несколько физических строк. Интерпретатор Python объединит следующие друг за другом строки, если последний символ строки — обратный слэш. В некоторых случаях это может оказаться полезным, но такого подхода следует избегать, потому что знак пробела, добавленный в конце строки, разрушит код и может привести к неожиданным последствиям.

Лучшее решение — заключить элементы в круглые скобки. Если интерпретатор Python встретит незакрытую круглую скобку в одной строке, он будет присоединять к ней следующие строки до тех пор, пока скобка не будет закрыта. То же поведение верно для фигурных и квадратных скобок.

[1] В соответствии с PEP 8 это значение равно 80 символам. Согласно другим источникам — 100, а в вашем случае это значение зависит от того, что говорит ваш начальник. Ха! Честно говоря, любой, кто использовал консоль для отладки кода в полевых условиях, быстро оценит ограничение в 80 символов (при котором строка в консоли не переносится) и на деле будет использовать 75–77, чтобы можно было увидеть нумерацию строк в Vi.

Плохой код	Хороший код

`french_insult = \`
`"Your mother was a hamster, and \`
`your father smelt of elderberries!"`
	`french_insult = (`
	`"Your mother was a hamster, and "`
	`"your father smelt of elderberries!"`
	`)`

`from some.deep.module.in.a.module \`
`import a_nice_function, \`
`another_nice_function, \`
`yet_another_nice_function`
	`from some.deep.module.in.a.module`
	`import (`
	`a_nice_function,`
	`another_nice_function,`
	`yet_another_nice_function`
	`)`

Однако зачастую необходимость разбивать длинные логические строки указывает на то, что вы пытаетесь выполнить слишком много действий за раз, что может навредить читаемости.

Идиомы

Несмотря на то что обычно существует всего один очевидный способ решить задачу, код Python, написанный с помощью идиом (*питонский* код), может поначалу казаться неочевидным для новичков (если только они не голландцы[1]). Поэтому вам необходимо освоить хорошие идиомы.

Распаковка

Если вы знаете длину списка или кортежа, можете присвоить имена их элементам с помощью распаковки. Поскольку вы можете указать количество разбиений строки для функций `split()` и `rsplit()`, правую сторону выражения присваивания можно разбить только один раз (например, на имя файла и расширение), а левая сторона может содержать оба места назначения одновременно, в правильном порядке. Например, так:

```
>>> filename, ext = "my_photo.orig.png".rsplit(".", 1)
>>> print(filename, "is a", ext, "file.")
my_photo.orig is a png file.
```

Вы можете задействовать распаковку для того, чтобы менять местами переменные:

```
a, b = b, a
```

Вложенная распаковка также работает:

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

[1] Обратитесь к 14-му пункту «Дзена Питона». Гвидо, наш BDFL, — голландец.

В Python 3 в PEP 3132 (https://www.python.org/dev/peps/pep-3132/) был представлен новый метод расширенной распаковки:

```
a, *rest = [1, 2, 3]
# a = 1, rest = [2, 3]

a, *middle, c = [1, 2, 3, 4]
# a = 1, middle = [2, 3], c = 4
```

Игнорирование значения

Если вам необходимо присвоить какое-то значение во время распаковки, но сама переменная не нужна, воспользуйтесь двойным подчеркиванием (__):

```
filename = 'foobar.txt'
basename, __, ext = filename.rpartition('.')
```

Многие руководства по стилю для Python рекомендуют использовать одинарное подчеркивание (_) для подобных переменных вместо двойного (__), о котором говорится здесь. Проблема в том, что одинарное подчеркивание зачастую применяется как псевдоним для функции gettext.gettext() и как интерактивное приглашение сохранить значение последней операции. Двойное подчеркивание выглядит точно так же прозрачно и почти так же удобно, снижает риск случайного переписывания переменной с именем «_» в обоих сценариях.

Создание списка длиной N, состоящего из одинаковых значений

Используйте оператор списка Python * для того, чтобы создать список, состоящий из одинаковых неизменяемых элементов:

```
>>> four_nones = [None] * 4
>>> print(four_nones)
[None, None, None, None]
```

Одинаковые объекты должны иметь одинаковые значения хэша. В документации к Python содержится более подробная информация.

Однако будьте осторожны при работе с изменяемыми объектами: поскольку списки изменяемы, оператор * создаст список, состоящий из *N* ссылок на него самого, и это вряд ли вас устроит. Поэтому используйте списковое включение:

Плохой код	**Хороший код**

`>>> four_lists = [[]] * 4`
`>>> four_lists[0].append("Ni")`
`>>> print(four_lists)`
`[['Ni'], ['Ni'], ['Ni'], ['Ni']]`
	`>>> four_lists = [[] for __ in range(4)]`
	`>>> four_lists[0].append("Ni")`
	`>>> print(four_lists)`
	`[['Ni'], [], [], []]`

Распространенная идиома для создания строк состоит в том, чтобы использовать функцию `str.join()` для пустой строки. Данная идиома может быть применена к спискам и кортежам:

```
>>> letters = ['s', 'p', 'a', 'm']
>>> word = ''.join(letters)
>>> print(word)
spam
```

Иногда требуется выполнить поиск по коллекции элементов. Изучим два варианта: списки и множества.

Для примера рассмотрим следующий код:

```
>>> x = list(('foo', 'foo', 'bar', 'baz'))
>>> y = set(('foo', 'foo', 'bar', 'baz'))
>>>
>>> print(x)
['foo', 'foo', 'bar', 'baz']
>>> print(y)
{'foo', 'bar', 'baz'}
>>>
>>> 'foo' in x True
>>> 'foo' in y True
```

Даже несмотря на то что обе булевых проверки на наличие в списке и множестве выглядят идентично, а `foo in y` учитывает тот факт, что множества (и словари) в Python являются хэш-таблицами[1], производительность для этих двух примеров будет различной. Python должен пройти по каждому элементу списка в поисках совпадения, на что уходит много времени (это заметно при увеличении размера коллекций). Но поиск ключей во множестве может быть выполнен быстро с помощью поиска по хэшу. Кроме того, множества и словари не могут содержать повторяющихся записей и идентичных ключей. Для получения более подробной информации поинтересуйтесь семинаром на эту тему на ресурсе Stack Overflow (http://stackoverflow.com/questions/5 13882).

Контексты с гарантией безопасности по исключениям

Зачастую блоки `try/finally` используются для управления ресурсами вроде файлов или блокировок потоков в случае генерации исключений. В PEP 343 (https://www.python.org/dev/peps/pep-0343/) представлены оператор `with` и протокол управления контекстом (в версиях 2.5 и выше) — идиома, позволяющая заменить блоки `try/finally` на более читаемый код. Протокол состоит из двух методов, `__enter__()`

[1] Кстати, именно поэтому только хэшируемые объекты можно хранить во множествах или использовать как ключи для словарей. Чтобы ваши объекты Python стали хэшируемыми, определите функцию-член object.__hash__(self), которая возвращает целое число.

и `__exit__()`, которые при реализации для объекта позволяют использовать этот объект в операторе `with`, например так:

```
>>> import threading
>>> some_lock = threading.Lock()
>>>
>>> with some_lock:
...     # Создать Землю 1, запустить ее на десять миллионов лет ...
...     print(
...         "Look at me: I design coastlines.\n"
...         "I got an award for Norway."
...     )
...
```

Раньше это выглядело бы так:

```
>>> import threading
>>> some_lock = threading.Lock()
>>>
>>> some_lock.acquire()
>>> try:
...     # Создать Землю 1, запустить ее на десять миллионов лет ...
...     print(
...         "Look at me: I design coastlines.\n"
...         "I got an award for Norway."
...     )
... finally:
...     some_lock.release()
```

Модуль стандартной библиотеки `contextlib` (https://docs.python.org/3/library/contextlib.html) предоставляет дополнительные инструменты, которые помогают преобразовать функции в менеджеры контекстов, навязать вызов метода `close()`, подавить исключения (в Python 3.4 и выше) и перенаправить стандартные потоки вывода и ошибок (в Python 3.4, 3.5 и выше). Рассмотрим пример использования функции `contextlib.closing()`:

```
>>> from contextlib import closing
>>> with closing(open("outfile.txt", "w")) as output:
...     output.write("Well, he's...he's, ah...probably pining for the fjords.")
...
56
```

Но поскольку методы `__enter__()` и `__exit__()` определены для объекта, который отвечает за ввод/вывод для файла[1], мы можем использовать это выражение непосредственно, не закрывая файл самостоятельно:

[1] В этом случае метод __exit__() вызовет метод обертки для ввода/вывода close(), чтобы закрыть дескриптор файла. Во многих системах имеется максимальное количество открытых дескрипторов файлов, и хороший тон — освобождать их по завершении работы.

```
>>> with open("outfile.txt", "w") as output:
    output.write(
       "PININ' for the FJORDS?!?!?!? "
       "What kind of talk is that?, look, why did he fall "
       "flat on his back the moment I got 'im home?\n"
    )
...
123
```

Распространенные подводные камни

По большей части Python — чистый и надежный язык. Однако некоторые ситуации могут быть непонятны для новичков: какие-то из них созданы намеренно, но все равно могут удивить, другие можно считать особенностями языка. В целом все, что продемонстрировано в этом подразделе, относится к неоднозначному поведению, которое может показаться странным на первый взгляд, но впоследствии выглядит разумным (когда вы узнаете о причинах).

Изменяемые аргументы по умолчанию

Наиболее частый сюрприз, с которым сталкиваются новые программисты Python, — это отношение Python к изменяемым аргументам по умолчанию в определениях функции.

Что вы написали:

```
def append_to(element, to=[]):
    to.append(element)
    return to
```

Чего вы ожидаете:

```
my_list = append_to(12)
print(my_list)
my_other_list = append_to(42)
print(my_other_list)
```

Новый список создается всякий раз, когда вызывается функция, если второй аргумент не предоставлен, поэтому результат работы функции выглядит так:

```
[12]
[42]
```

Что происходит на самом деле:

```
[12]
[12, 42]
```

Новый список создается при определении функции, он же используется в момент каждого последующего вызова: аргументы по умолчанию в Python оцениваются

при определении функции, а не при каждом ее вызове (как это происходит, например, в Ruby).

Это означает, что если вы используете изменяемый по умолчанию аргумент и измените его, то он изменится для всех последующих вызовов этой функции.

Что вам нужно сделать вместо этого? Создавайте новый объект при каждом вызове функции, используя аргумент по умолчанию, чтобы показать, что аргумент не был передан (в качестве такого значения подойдет `None`):

```
def append_to(element, to=None):
    if to is None:
        to = []
    to.append(element)
    return to
```

Когда подводный камень вовсе не подводный камень. Иногда вы можете намеренно задействовать (то есть использовать в качестве нормального варианта поведения) этот подводный камень, чтобы сохранять состояние между вызовами функции. Зачастую это делается при написании функции кэширования (которая сохраняет результаты в памяти), например:

```
def time_consuming_function(x, y, cache={}):
    args = (x, y)
    if args in cache:
        return cache[args]
    # В противном случае функция работает с аргументами в первый раз.
    # Выполняем сложную операцию...
    cache[args] = result
    return result
```

Замыкания с поздним связыванием

Еще один распространенный источник путаницы — способ связывания переменных в замыканиях (или в окружающей глобальной области видимости).

Что вы написали:

```
def create_multipliers():
    return [lambda x : i * x for i in range(5)]
```

Чего вы ожидаете:

```
for multiplier in create_multipliers():
    print(multiplier(2), end=" ... ")
print()
```

Список, содержащий пять функций, каждая из них имеет собственную замкнутую переменную `i`, которая умножается на их аргумент, что приводит к получению следующего результата:

```
0 ... 2 ... 4 ... 6 ... 8 ...
```

Что происходит на самом деле:

```
8 ... 8 ... 8 ... 8 ... 8 ...
```

Создаются пять функций, все они умножают `x` на `4`. Почему? В Python замыкания имеют *позднее связывание*. Это говорит о том, что значения переменных, использованных в замыканиях, определяются в момент вызова внутренней функции.

В нашем примере, когда вызывается *любая* из возвращенных функций, значение переменной `i` определяется с помощью окружающей области видимости в момент вызова. К этому моменту цикл завершает свою работу и `i` получает итоговое значение `4`.

Особенно неудобно то, что вам может показаться, будто ошибка как-то связана с лямбда-выражениями (https://docs.python.org/3/tutorial/controlflow.html#lambda-expressions). Функции, создаваемые с помощью лямбда-выражений, не отличаются от других. Фактически то же самое поведение проявляется и при использовании самого обычного `def`:

```
def create_multipliers():
    multipliers = []
    for i in range(5):
        def multiplier(x):
            return i * x
        multipliers.append(multiplier)
    return multipliers
```

Что вам нужно сделать вместо этого? Наиболее общее решение, возможно, станет «костылем» — временным вариантом устранения проблемы. Из-за уже упомянутого поведения Python, связанного с определением аргументов по умолчанию для функций (см. предыдущий пункт «Изменяемые аргументы функций»), вы можете создать замыкание, которое немедленно связывается со своими аргументами с помощью аргумента по умолчанию:

```
def create_multipliers():
    return [lambda x, i=i : i * x for i in range(5)]
```

Помимо этого вы можете использовать функцию `functools.partial()`:

```
from functools import partial
from operator import mul
def create_multipliers():
    return [partial(mul, i) for i in range(5)]
```

Когда подводный камень вовсе не подводный камень. Иногда нужно, чтобы замыкания вели себя подобным образом. Позднее связывание может быть полезным во многих ситуациях (например, в проекте Diamond, см. пункт «Пример использования замыкания (когда подводный камень вовсе не подводный камень)» на с. 136). Наличие уникальных функций в циклах, к сожалению, может привести к сбоям.

Структурируем проект

Под *структурированием* мы понимаем решения, которые вы принимаете по поводу функционирования вашего проекта. Его цель состоит в использовании возможностей Python для создания чистого и эффективного кода. На практике это означает, что логика и зависимости в коде и структуре файлов и каталогов прозрачны.

По какому принципу функции должны размещаться в модулях? Как данные перемещаются по проекту? Какие функции могут быть сгруппированы и изолированы? Отвечая на эти вопросы, вы можете запланировать, как будет выглядеть ваш конечный продукт.

В книге *Python Cookbook* есть глава, посвященная модулям и пакетам (http://bit.ly/python-cookbook-ch10), в которой подробно описывается, как работают выражения `__import__` и упаковка. Цель этого раздела — осветить основные аспекты системы модулей и импортирования Python, необходимые для структурирования ваших проектов. Далее мы рассмотрим разные подходы к сборке кода, который легко будет расширять и тестировать.

Благодаря тому, как в Python налажен процесс импортирования и разбиения на модули, структурировать проект довольно просто: существует всего несколько ограничений, модель для импортирования также нетрудно освоить. Поэтому перед вами стоит исключительно архитектурная задача — создать различные части проекта и продумать их взаимодействие.

Модули

Модуль — это один из основных уровней абстракции в Python. Уровни абстракции позволяют программисту разбивать код на части, которые содержат связанные данные и функциональность.

Например, если один уровень проекта предназначен для взаимодействия с пользователем, а другой обрабатывает данные на низком уровне, наиболее логичным способом разделения этих двух слоев является размещение всей функциональности, связанной со взаимодействием, в одном файле, а всех низкоуровневых операций — в другом. Такая группировка разметит их в два разных модуля. Файл для взаимодействия затем импортирует файл для низкоуровневой обработки с помощью выражения `import module` или `from module import attribute`.

Как только вы пустите в ход выражение `import`, вы начнете пользоваться модулями. Модули могут быть либо встроенными (вроде `os` и `sys`), либо сторонними пакетами, установленными в среде (вроде Requests или NumPy), либо внутренними модулями проекта.

Далее показан пример некоторых выражений `import` (подтверждается, что импортированный модуль является объектом Python со своим типом данных):

```
>>> import sys  # built-in module
>>> import matplotlib.pyplot as plt  # сторонний модуль
>>>
>>> import mymodule as mod  # внутренний модуль проекта
>>>
>>> print(type(sys), type(plt), type(mod))
<class 'module'> <class 'module'> <class 'module'>
```

В соответствии с руководством по стилю кода (https://www.python.org/dev/peps/pep-0008/) присваивайте модулям короткие имена, которые начинаются со строчной буквы. И убедитесь, что не использовали специальные символы вроде точки (`.`) или вопросительного знака (`?`), поскольку это может нарушить вид Python для модулей. Поэтому вам следует избегать имен файла вроде `my.spam.py`[1] (Python попытается найти файл `spam.py` в каталоге с именем `my`, а это неверно). В документации Python (http://docs.python.org/tutorial/modules.html#packages) более подробно описывается нотация с точкой.

Импортирование модулей. Помимо следования некоторым ограничениям в именовании, для использования файла Python в качестве модуля не требуется больше ничего особенного. Однако понимать механизм импортирования будет нелишним. Во-первых, выражение `import modu` начнет искать определение `modu` в файле с именем `modu.py` в том же каталоге, где находится и вызывающая сторона, если такой файл существует. При неудаче интерпретатор Python будет рекурсивно искать файл `modu.py` в пути поиска Python (https://docs.python.org/2/library/sys.html#sys.path) и сгенерирует исключение `ImportError`, если не найдет. Путь поиска зависит от платформы и включает в себя определенные пользователем или системой каталоги, указанные в переменной среды `$PYTHONPATH` (или `%PYTHONPATH%` в Windows). Ее можно просмотреть или изменить в сессии Python:

```
import sys
>>> sys.path
[ '', '/current/absolute/path', 'etc']
# Реальный список содержит каждый путь, где выполняется поиск,
# когда вы импортируете библиотеки в Python в том порядке,
# в котором они проверяются.
```

Как только файл `modu.py` будет найден, интерпретатор Python запустит модуль в ограниченной области видимости. Любое выражение верхнего уровня в файле `modu.py` будет выполнено, включая другие выражения импорта, если таковые существуют. Определения функций и классов хранятся в словаре модуля. Наконец, переменные функции и классы модуля будут доступны вызывающей стороне с помощью *пространства имен* модуля — основной концепции программирования, которая особенно эффективна в Python. Пространства имен предоставляют область

[1] Если хотите, можете назвать файл my_spam.py, но даже нашим другом — нижним подчеркиванием — не следует злоупотреблять в именах модулей (нижнее подчеркивание наводит на мысль, что перед вами имя переменной).

видимости, содержащую именованные атрибуты, которые видны друг другу, но к ним нельзя получить доступ из-за пределов пространства имен.

Во многих языках директива заставляет препроцессор, по сути, скопировать содержимое включаемого файла в код вызывающей стороны. В Python все происходит иначе: включаемый код изолируется в пространстве имен модуля. Результатом выполнения выражения `import modu` станет объект модуля с именем `modu`, который будет находиться в глобальном пространстве имен, его атрибуты будут доступны с помощью точечной нотации. Например `modu.sqrt` — это объект `sqrt`, определенный внутри файла `modu.py`. Это означает, что вам, как правило, не нужно волноваться о том, что включаемый код может делать что-то нежелательное, к примеру переопределять существующую функцию с тем же именем.

Инструменты для пространств имен

Функции `dir()`, `globals()` и `locals()` помогают быстро исследовать пространства имен:

- `dir(object)` возвращает список атрибутов, к которым объект может получить доступ;
- `globals()` возвращает словарь атрибутов, находящихся в данный момент в глобальном пространстве имен, а также их значения;
- `locals()` возвращает словарь атрибутов в текущем локальном пространстве имен (например, внутри функции), а также их значения.

Для получения более подробной информации обратитесь к разделу Data model официальной документации Python (https://docs.python.org/3/reference/datamodel.html).

Вы можете симулировать более привычное поведение, используя специальный синтаксис в выражении `import: from modu import *`. Однако это, как правило, считается признаком плохого тона: наличие конструкции `import *` усложняет чтение кода, делает зависимости более связанными и может затереть (перезаписать) существующие определенные объекты новыми описаниями из импортированного модуля.

Нотация `from modu import func` — это способ импортировать только необходимые вам атрибуты в глобальное пространство имен. Она гораздо безопаснее нотации `from modu import *`, поскольку явно показывает, что именно импортируется в глобальное пространство имен. Единственное ее преимущество перед более простой нотацией `import modu` в том, что она сэкономит вам немного времени.

В табл. 4.1 сравниваются разные способы импортирования определений из других модулей.

Таблица 4.1. Разные способы импортировать определения из модулей

Очень плохой код (непонятный для читателя)	Код получше (здесь понятно, какие имена находятся в глобальном пространстве имен)	Лучший код (сразу понятно, откуда появился тот или иной атрибут)
from modu import *	from modu import sqrt	import modu
x = sqrt(4)	x = sqrt(4)	x = modu.sqrt(4)
from modu import sqrt	from modu import sqrt	from modu import sqrt

Как упоминается в разделе «Стиль кода» в начале этой главы, читаемость — одна из основных особенностей Python. Читаемый код не содержит бесполезного текста. Но не следует максимально его сокращать в угоду краткости. Явно указывая, откуда появился тот или иной класс или функция, как в случае идиомы `modu.func()`, вы повышаете читаемость кода и степень его понимания.

Структура — это главное

Несмотря на то что вы можете структурировать проект так, как вам нравится, следует избегать некоторых ошибок.

- *Большое количество запутанных циклических зависимостей.* Если для ваших классов `Table` и `Chair` из файла `furn.py` нужно импортировать класс `Carpenter` из файла `workers.py` (чтобы ответить на вопрос `table.is_done_by()` («произведены кем?»)) и если для класса `Carpenter` нужно импортировать классы `Table` и `Chair` (чтобы ответить на вопрос `carpenter.what_do()` («что производит?»)), у вас имеется циклическая зависимость: файл `furn.py` зависит от файла `workers.py`, который зависит от файла `furn.py`. В таком случае вам нужно использовать выражение `import` внутри методов, дабы избежать исключения `ImportError`.
- *Скрытое связывание.* После каждого изменения в реализации класса `Table` вдруг перестают работать 20 несвязанных с ним тестов, поскольку это нарушает реализацию класса `Carpenter`. Это требует проведения аккуратных изменений для того, чтобы к ним адаптироваться, и означает, что в своем коде класса `Carpenter` вы делаете слишком много предположений о классе `Table`.
- *Избыточное использование глобального состояния или контекста.* Вместо явной передачи данных (высота, ширина, тип, древесина) друг другу классы `Table` и `Carpenter` полагаются на глобальные переменные, которые модифицируются на лету разными агентами. Вам придется перебрать все объекты, имеющие доступ к этим глобальным переменным, чтобы понять, почему прямоугольный стол стал квадратным, и обнаружить, что это сделал код, который отвечает за работу шаблонов.
- *Спагетти-код.* Вложенные условия `if`, расположенные на нескольких страницах подряд, и циклы `for`, содержащие большое количество скопированного

кода процедур и плохо отформатированные, называются *спагетти-кодом*. Поскольку в Python отступы имеют смысл (одна из его наиболее противоречивых особенностей), написать такой код будет сложно и вы вряд ли будете часто с ним сталкиваться.

- *Равиоли-код*. Такой код в Python встретить более вероятно, чем спагетти-код. *Равиоли-код* состоит из сотен небольших логических фрагментов, зачастую классов или объектов, которые не имеют хорошей структуры. Если вы не можете вспомнить, нужны ли вам для выполнения текущей задачи классы `FurnitureTable`, `AssetTable`, `Table` или даже `TableNew`, то, скорее всего, работаете с равиоли-кодом.

Упаковка

Python предоставляет довольно понятную систему упаковки, которая расширяет механизм модулей так, что он начинает работать с каталогами.

Любой каталог, содержащий файл `__init__.py`, считается пакетом Python. Каталог высшего уровня, в котором находится файл `__init__.py`, является *корневым пакетом*[1]. Разные модули пакетов импортируются аналогично простым модулям, но файл `__init__.py` при этом будет использован для сбора всех описаний на уровне пакета.

Файл `modu.py`, находящийся в каталоге `pack/`, импортируется с помощью выражения `import pack.modu`. Интерпретатор выполнит поиск файла `__init__.py` в `pack` и запустит все его выражения верхнего уровня. Затем выполнит поиск файла с именем `pack/modu.py` и запустит все его выражения верхнего уровня. После этих операций любая переменная, функция или класс, определенные в файле `modu.py`, будут доступны пространству имен `pack.modu`.

Распространенная проблема заключается в том, что файлы `__init__.py` содержат слишком много кода. Когда сложность проекта повышается, в структуре каталогов могут появляться подпакеты и подподпакеты. В этом случае импортирование одного элемента из подподпакета потребует запуска всех файлов `__init__.py`, встреченных в дереве на пути к искомому.

Признаком хорошего тона является поддержание файла `__init__.py` пустым, когда модули и подпакеты пакета не имеют общего кода. Проекты HowDoI и Diamond, использованные в качестве примеров в следующем разделе, не содержат кода

[1] Благодаря PEP 420 (https://www.python.org/dev/peps/pep-0420/), который был реализован в Python 3.3, существует альтернативный корневой пакет — пакет пространства имен. Такие пакеты не должны содержать файл __init__.py, могут быть разбиты по нескольким каталогам sys.path. Python соберет все фрагменты воедино и представит их пользователю как один пакет.

в файлах `__init__.py`, помимо номеров версий. В проектах Tablib, Requests и Flask в этом файле есть строка документации верхнего уровня и выражения импорта, предоставляющие API каждого проекта. Проект Werkzeug также предоставляет API верхнего уровня, но делает это с помощью ленивой загрузки (дополнительного кода, который добавляет содержимое в пространство имен, только когда тот используется, что ускоряет работу исходного выражения импорта).

Наконец, для импортирования глубоких вложенных пакетов доступен удобный синтаксис: `import very.deep.module as mod`. Это позволяет использовать слово `mod` на месте избыточной конструкции `very.deep.module`.

Объектно-ориентированное программирование

Python иногда описывается как объектно-ориентированный язык. Это может вносить путаницу, поэтому давайте проясним данный вопрос.

В Python все элементы являются объектами и могут быть обработаны как объекты. Именно это мы имеем в виду, когда говорим, что функции являются объектами первого класса. Функции, классы, строки и даже типы считаются в Python объектами: все они имеют тип, их можно передать как аргументы функций, они могут иметь методы и свойства. С этой точки зрения Python действительно объектно-ориентированный язык.

Однако, в отличие от Java, в Python парадигма объектно-ориентированного программирования не будет основной. Проект, написанный на Python, вполне может быть не объектно-ориентированным, то есть в нем не будут использоваться (или будут, но в небольших количествах) определения классов, наследование классов или другие механизмы, характерные для объектно-ориентированного программирования. Для питонистов эта функциональность *доступна*, но *необязательна*. Более того, как вы могли увидеть в подразделе «Модули» текущего раздела, способ, с помощью которого Python обрабатывает модули и пространства имен, дает разработчику возможность гарантировать инкапсуляцию и разделение между абстрактными уровнями — наиболее распространенную причину использования парадигмы объектно-ориентированного программирования — *без наличия* классов.

Защитники функционального программирования (парадигма, которая в своей чистейшей форме не имеет операторов присваивания и побочных эффектов и вызывает функции одну за другой, чтобы выполнить задачу) могут утверждать: из-за того, что функция выполняет разную работу в зависимости от состояния системы (например, от глобальной переменной, которая указывает, вошел ли пользователь под своей учетной записью), могут возникать ошибки и путаница. В Python (несмотря на то что он не является чисто функциональным языком) имеются инструменты, которые позволяют заниматься функциональным программированием (http://bit.ly/functional-programming-python). Мы можем ограничить применение

пользовательских классов до ситуаций, когда понадобится объединить состояние и функциональность.

В некоторых архитектурах, обычно в веб-приложениях, создается несколько процессов Python для того, чтобы реагировать на внешние запросы, которые могут происходить одновременно. В этом случае сохранение состояния созданных объектов (означает хранение статичной информации о мире) может привести к *состоянию гонки*. Этот термин употребляется при описании ситуации, когда в какой-то момент между инициализацией состояния объекта (которая в Python выполняется с помощью метода `Class.__init__()`) и использованием его состояния с помощью одного из методов состояние мира изменилось.

Например, запрос может загрузить предмет в память и затем пометить, что он добавлен в корзину пользователя. Если другой запрос в то же время «продаст» такой же предмет другому человеку, может случиться, что продажа на самом деле произойдет после того, как первая сессия добавит предмет (затем мы попытаемся продать предмет, который уже помечен как проданный). Подобные проблемы приводят к тому, что многие предпочитают функции, не сохраняющие состояние.

Мы дадим следующую рекомендацию: при работе с кодом, полагающимся на некий устойчивый контекст или глобальное состояние (как и многие веб-приложения), используйте функции и процедуры, которые привнесут минимальное количество неявных контекстов и побочных эффектов. Неявный контекст функции создается из любых глобальных переменных и элементов на уровне сохраняемости, к которым можно получить доступ из функции. *Побочные эффекты* — это изменения, которые функция вносит в свой неявный контекст. Если функция сохраняет или удаляет данные в глобальной переменной или на уровне сохраняемости, можно сказать, что она имеет побочные эффекты.

Пользовательские классы в Python необходимо применять для того, чтобы аккуратно изолировать функции, имеющие контексты и побочные эффекты, от функций, которые имеют логику (называются *чистыми функциями*). Чистые функции всегда определены: учитывая фиксированные входные данные, результат их работы неизменен, потому что они не зависят от контекста и не имеют побочных эффектов. Функция `print()`, например, не является чистой, поскольку ничего не возвращает, а записывает данные в стандартный поток ввода-вывода как побочный эффект.

Рассмотрим преимущества чистых функций:

- их проще изменить или заменить, если нужно выполнить рефакторинг;
- их проще тестировать с помощью юнит-тестов, не нужно выполнять сложную настройку контекста и очищать данные после ее работы;
- ими проще манипулировать, их легче декорировать (к этой теме мы сейчас вернемся) и передавать.

В итоге для некоторых инфраструктур чистые функции выступают более эффективными строительными блоками, чем классы или объекты, поскольку не имеют контекста и побочных эффектов. В качестве примера рассмотрим функции ввода-вывода, связанные с каждым форматом файла в библиотеке Tablib (`tablib/formats/*.py` — мы опишем Tablib в следующей главе). Они являются чистыми функциями, а не частью класса, поскольку лишь считывают данные из отдельного объекта типа `Dataset`, в котором хранятся, либо записывают объект типа `Dataset` в файл. Но объект типа `Session` в библиотеке Requests (ее мы также рассмотрим в следующей главе) — это класс, поскольку он должен сохранять cookies и информацию об аутентификации, которая может пригодиться при обмене данными в ходе сессии HTTP.

Объектно-ориентированное программирование — полезная и даже необходимая парадигма программирования во многих случаях, например при разработке графических приложений для десктопа или игр, где вы можете манипулировать объектами (окнами, кнопками, аватарами, машинами), которые долго живут в памяти компьютера; является одной из причин использовать объектно-реляционное отображение, которое соотносит строки базы данных с объектами в коде. Этот вопрос рассматривается в разделе «Библиотеки для работы с базами данных» главы 11.

Декораторы

Декораторы были добавлены в Python в версии 2.4, определены и рассмотрены в PEP 318 (https://www.python.org/dev/peps/pep-0318/). Декоратор — это функция или метод класса, которые оборачивают (или декорируют) другую функцию или метод. Декорированная функция или метод заменят оригинал. Поскольку функции являются объектами первого класса в Python, декорирование можно выполнить вручную, но все же более предпочтителен синтаксис `@decorator`. Рассмотрим пример использования декоратора:

```
>>> def foo():
...     print("I am inside foo.")
...
...
...
>>> import logging
>>> logging.basicConfig()
>>>
>>> def logged(func, *args, **kwargs):
...     logger = logging.getLogger()
...     def new_func(*args, **kwargs):
...         logger.debug("calling {} with args {} and kwargs {}".format(
...                      func.__name__, args, kwargs))
...         return func(*args, **kwargs)
...     return new_func
```

```
...
>>>
>>>
... @logged
... def bar():
...     print("I am inside bar.")
...
>>> logging.getLogger().setLevel(logging.DEBUG)
>>> bar()
DEBUG:root:calling bar with args () and kwargs {}
I am inside bar.
>>> foo()
I am inside foo.
```

Этот механизм подойдет, чтобы изолировать основную логику функции или метода. Примером задачи, для которой нужно использовать декорирование, можно назвать запоминание или кэширование: вы хотите сохранить результат дорогой функции в таблице и использовать его вместо того, чтобы выполнять повторные вычисления. Очевидно, это не является частью логики функции. В PEP 3129 (https://www.python.org/dev/peps/pep-3129/), начиная с Python 3, декораторы также можно применять к классам.

Динамическая типизация

Python — динамически типизированный язык (в противоположность статически типизированным). Это означает, что переменные не имеют фиксированного типа. Переменные реализуются как указатели на объект, что дает возможность задать сначала значение `42`, затем значение `thanks for all the fish`, а потом установить в качестве значения функцию.

Динамическая типизация, используемая в Python, зачастую считается недостатком, поскольку может привести к сложностям и появлению кода, для которого сложно выполнять отладку: если именованный объект может иметь в качестве значения множество разных вещей, разработчик поддерживающий код, должен отслеживать это имя в коде, чтобы убедиться, что оно не получило неуместное значение. В табл. 4.2 перечислены правила хорошего и плохого тона при именовании.

Таблица 4.2. Правила хорошего и плохого тона при задании имен

Совет	Плохой код	Хороший код

Используйте короткие функции или методы, чтобы снизить риск указания одного имени для двух несвязанных объектов	`a = 1`
	`a = 'answer is {}'.format(a)`
		`def get_answer(a):`
		`  return 'answer is {}'.format(a)`
		`a = get_answer(1)`

Продолжение ⇗

Таблица 4.2 *(продолжение)*

Совет	Плохой код	Хороший код

Используйте разные имена для связанных элементов, если они имеют разные типы	`# Строка ...`
	`items = 'a b c d'`
	`# А теперь список`
	`items = items.`
	`split(' ')`
	`# А теперь множество`
	`items = set(items)`
		`items_string = 'a b c d'`
		`items_list = items.`
		`split(' ')`
		`items = set(items_list)`

Повторное использование имен не повышает эффективность: операция присваивания все равно создаст новый объект. При росте сложности, когда операции присваивания разделены другими строками кода, сложно определить тип переменной.

В некоторых видах программирования, включая функциональное, не рекомендуется пользоваться возможностью повторного присваивания значения переменным. В Java вы можете указать, что переменная всегда будет содержать одно и то же значение после присваивания, с помощью ключевого слова `final`. В Python такого ключевого слова нет (это шло бы вразрез с его философией). Но присваивание значения переменной всего один раз может быть признаком дисциплинированности. Это помогает поддержать концепцию изменяемых и неизменяемых типов.

Pylint (https://www.pylint.org/) предупредит вас, если вы попытаетесь присвоить переменной, уже содержащей значение одного типа, значение другого типа.

Изменяемые и неизменяемые типы

В Python имеются два типа встроенных или определяемых пользователем[1] типов:

```
# Списки можно изменять
my_list = [1, 2, 3]
my_list[0] = 4
print my_list  # [4, 2, 3] <- тот же список, измененный.
# Целые числа изменять нельзя
x = 6
x = x + 1  # Новое значение x занимает другое место в памяти.
```

- *Изменяемые типы*. Позволяют изменять содержимое объекта на месте. Примерами могут стать списки и словари, которые имеют изменяющие методы вроде `list.append()` или `dict.pop()` и могут быть модифицированы на месте.

[1] Инструкции по созданию собственных типов с помощью C предоставлены в документации Python по адресу https://docs.python.org/3/extending/newtypes.html.

- *Неизменяемые типы*. Не предоставляют методов для изменения их содержимого. Например, переменная `x` со значением `6` не имеет метода для инкремента. Для того чтобы вычислить значение выражения `x + 1`, нужно создать другую целочисленную переменную и дать ей имя.

Одно из последствий такого поведения — объекты изменяемых типов не могут быть использованы как ключи для словаря, ведь если их значение изменится, то изменится и его хэш (словари используют хэширование[1] для хранения ключей). Неизменяемым эквивалентом списка является кортеж. Он создается добавлением круглых скобок, например `(1, 2)`. Кортеж нельзя изменить на месте, поэтому его можно использовать как ключ словаря.

Правильное применение изменяемых типов для объектов, которые по задумке должны изменяться (например, `my_list = [1, 2, 3]`), и неизменяемых типов для объектов, которые по задумке должны иметь фиксированное значение (например, `islington_phone = ("220", "7946", "0347")`), поможет другим разработчикам понять код.

В Python строки неизменяемы и это может удивить новичков. Попытка изменить строку вызовет ошибку:

```
>>> s = "I'm not mutable"
>>> s[1:7] = " am"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

Это означает, что при создании строки по частям гораздо эффективнее собрать все части в список, поскольку его можно изменять, а затем объединить их. Кроме того, в Python предусмотрены *списковые включения*, которые предоставляют простой синтаксис для итерирования по входным данным для создания списка. В табл. 4.3 приведены способы создания строки из итерабельного объекта.

Таблица 4.3. Способы конкатенации строки

Плохой	Хороший	Лучший

`>>> s = ""`
`>>> for c in (97, 98, 98):`
`...  s += unichr(c)`
`...`
`>>> print(s)`
`abc`
	`>>> s = []`
	`>>> for c in (97, 98, 99):`
	`...  s.append(unichr(c))`
	`...`
	`>>> print("".join(s))`
	`abc`
		`>>> r = (97, 98, 99)`
		`>>> s = [unichr(c) for`
		`  c in r]`
		`>>> print("".join(s))`
		`abc`

[1] Пример простого алгоритма хэширования — преобразование байтов объекта в целое число и взятие его суммы по какому-нибудь модулю. memcached (http://www.memcached.org/) распределяет ключи между несколькими компьютерами именно так.

На главной странице Python (https://www.python.org/doc/essays/list2str/) вы можете найти обсуждение подобной оптимизации.

Наконец, если количество элементов конкатенации известно, добавить строку будет проще (и очевиднее), чем создавать список элементов только для того, чтобы вызвать функцию `"".join()`.

Все следующие варианты форматирования для определения переменной `cheese` делают одно и то же[1]:

```
>>> adj = "Red"
>>> noun = "Leicester"
>>>
>>> cheese = "%s %s" % (adj, noun)  # Этот стиль устарел (PEP 3101)
>>> cheese = "{} {}".format(adj, noun)  # Возможно начиная с Python 3.1
>>> cheese = "{0} {1}".format(adj, noun)  # Числа можно использовать повторно
>>> cheese = "{adj} {noun}".format(adj=adj, noun=noun)  # Этот стиль – лучший
>>> print(cheese)
Red Leicester
```

Зависимости, получаемые от третьей стороны

Пакет, который использует *зависимости, получаемые от третьей стороны*, содержит внешние зависимости (сторонние библиотеки) внутри своего исходного кода, зачастую внутри каталога с именем `vendor` или `packages`. По адресу http://bit.ly/onvendorizing вы можете прочесть весьма полезную статью, в которой перечисляются основные причины, почему владелец пакета может воспользоваться зависимостями третьей стороны (в основном для того, чтобы избежать проблем с совместимостью), а также рассматриваются альтернативные подходы.

Однако можно достичь консенсуса: почти во всех случаях лучше всего держать зависимости отдельно друг от друга, поскольку это добавляет ненужное содержимое (зачастую мегабайты дополнительного кода) в репозиторий. Виртуальные среды, использованные в сочетании с файлами `setup.py` (предпочтительно, особенно если пакет является библиотекой) или `requirements.txt` (при использовании переопределит зависимости в файле `setup.py` в случае конфликтов), могут ограничить зависимости набором рабочих версий.

Если этих вариантов недостаточно, можно связаться с владельцем зависимости, чтобы решить проблему, обновив его пакет (например, ваша библиотека может зависеть от выходящего релиза его пакета или вам нужна новая функциональность).

---

[1] Должны признать: хотя в PEP 3101 (https://www.python.org/dev/peps/pep-3101/) форматирование с использованием знака % (%s, %d, %f) считается устаревшим, большинство разработчиков старой школы все еще используют его. В PEP 460 (https://www.python.org/dev/peps/pep-0460/) был представлен такой же метод форматирования байтов или объектов bytearray.

Эти изменения, скорее всего, пойдут на пользу всему сообществу. Однако здесь имеется и подводный камень: если вы отправите запрос на включение больших изменений, вам, возможно, придется поддерживать эти изменения по мере появления дальнейших предложений и запросов (по этой причине в проектах Tablib и Requests несколько зависимостей получены от третьей стороны). По мере полного перехода сообщества на Python 3 мы надеемся, что проблемных областей станет меньше.

Тестирование вашего кода

Тестировать код очень важно. Ведь люди будут использовать только такой проект, который на самом деле работает.

Модули `doctest` и `unittest` впервые появились в версии Python 2.1 (выпущена в 2001 году), поддерживая *разработку через тестирование* (test-driven development, TDD): разработчик сначала пишет тесты, которые определяют основную задачу и узкие места функции, а затем — функцию, которая проходит эти тесты. С тех пор TDD стали чаще использовать в бизнес-проектах и проектах с открытым исходным кодом — практиковаться в написании кода теста и параллельно самой функции довольно полезно. Если пользоваться этим методом с умом, он поможет вам четко определить предназначение своего кода и создать развернутую модульную структуру.

Советы по тестированию

Тест — это самый объемный фрагмент кода, который автостопщик может написать. Приведем несколько советов.

Тестируйте что-то одно за раз. Юнит-тест должен концентрироваться на небольшом фрагменте функциональности и доказывать, что все работает, как требуется.

Независимость императивна. Каждый юнит-тест должен быть полностью независимым: его можно запустить как отдельно, так и внутри набора тестов без учета того, в каком порядке они вызываются. Из этого правила следует, что для каждого теста нужно загрузить свежий набор данных, а после его выполнения провести очистку (обычно с помощью методов `setUp()` и `tearDown()`).

Точность лучше простоты. Используйте длинные описательные имена для функций теста. Это правило отличается от правила для рабочего кода, где предпочтительны короткие имена. Причина в том, что функции никогда не вызываются явно. В рабочем коде допускается использование имен `square()` или даже `sqr()`, но в коде теста у вас должны быть имена вроде `test_square_of_number_2()` или `test_square_negative_number()`. Эти имена функций будут выведены, когда тест даст сбой, они должны быть максимально описательными.

Скорость имеет значение. Старайтесь писать тесты, которые работают быстро. Если для того, чтобы тест отработал, нужно несколько миллисекунд, разработка

будет замедлена или тесты будут запускаться не так часто, как вам бы этого хотелось. В некоторых случаях тесты не могут быть быстрыми, поскольку для их работы требуется сложная структура данных, которая должна подгружаться каждый раз, когда запускается тест. Держите подобные тесты в отдельном наборе, который запускается какой-нибудь задачей по графику, а остальные тесты запускайте так часто, как вам это нужно.

RTMF (Read the manual, friend! — «Читай руководство, друг!»). Изучайте свои инструменты, чтобы знать, как запустить отдельный тест или набор тестов. При разработке функции внутри модуля почаще запускайте тесты для нее, в идеале всякий раз, когда вы сохраняете код.

Тестируйте все в начале работы и затем опять тестируйте по ее завершении. Всегда запускайте полный набор тестов перед тем, как писать код, и по завершении работы. Это позволит убедиться, что вы ничего «не сломали» в остальной части кода.

Автоматические функции перехвата для системы управления версиями фантастически хороши. Реализовать функцию перехвата, которая запускает все тесты перед тем, как отправить код в общий репозиторий, — хорошая идея. Вы можете непосредственно добавлять функции перехвата в вашу систему контроля версий, некоторые IDE предоставляют способы сделать это с помощью их собственных сред. Далее приведены ссылки на документацию к популярным системам контроля версий, в которой содержится информация о том, как это реализовать:

- GitHub (https://developer.github.com/webhooks/);
- Mercurial (http://bit.ly/mercurial-handling-repo);
- Subversion (http://bit.ly/svn-repo-hook).

Напишите тест, если хотите сделать перерыв. Если вы остановились на середине сессии разработки и вам нужно прервать работу, можете написать неработающий тест, который связан с тем, что вы планируете реализовать. По возвращении к работе у вас будет указатель на то место, где вы остановились (вы сможете приступить быстрее).

В случае неопределенности выполните отладку для теста. Первый шаг отладки кода — написание еще одного теста, который указывает на ошибку. Несмотря на то что это не всегда можно сделать, тесты, отлавливающие ошибки, являются наиболее ценными фрагментами кода вашего проекта.

Если тест сложно объяснить, то желаем вам удачи в поиске коллег. Если что-то идет не так или что-то нужно изменить и для вашего кода написано множество тестов, вы или другие сотрудники, работающие над проектом, будете полагаться на набор тестов для решения проблемы или изменения поведения. Поэтому код теста должен быть читаемым на том же уровне (или даже *больше*), чем рабочий код. Юнит-тест, чье предназначение неясно, не принесет большой пользы.

Если тест просто объяснить, он почти всегда хорош. Код теста можно использовать в качестве руководства для новых разработчиков. Если другим людям нужно работать с базой кода, запуск и чтение соответствующих тестов — это лучшее, что они могут сделать. Они обнаружат (по крайней мере должны обнаружить) проблемные места, вызывающие больше всего трудностей, а также пограничные случаи. Если им нужно добавить какую-то функциональность, в первую очередь следует добавить тест (это гарантирует ее появление).

Не паникуйте! Это же ПО с открытым исходным кодом! Вас поддержит весь мир.

Основы тестирования

В этом разделе приводятся основы тестирования, чтобы у вас было представление о доступных вариантах, и примеры из проектов Python, которые мы рассмотрим в главе 5. Есть целая книга, посвященная TDD в Python, мы не хотим переписывать ее здесь. Она называется *Test-Driven Development with Python* (издательство O'Reilly).

unittest

unittest — это тестовый модуль стандартной библиотеки Python, готовый к работе сразу после установки. Его API будет знаком всем, кто пользовался любым из этих инструментов — JUnit (Java)/nUnit (.NET)/CppUnit (C/C++).

Создать тест в этом модуле можно путем создания подкласса для `unittest.TestCase`. В этом примере функция тестирования определяется как новый метод в `MyTest`:

```
# test_example.py
import unittest
def fun(x):
    return x + 1
class MyTest(unittest.TestCase):
    def test_that_fun_adds_one(self):
        self.assertEqual(fun(3), 4)
class MySecondTest(unittest.TestCase):
    def test_that_fun_fails_when_not_adding_number(self):
        self.assertRaises(TypeError, fun, "multiply six by nine")
```

Методы теста должны начинаться со строки test — иначе они не запустятся. Тестовые модули должны следовать шаблону test*.py по умолчанию, но могут соответствовать любому шаблону, который вы передадите с помощью аргумента с ключевым словом pattern в командной строке.

Для того чтобы запустить все тесты в `TestClass`, откройте терминальную оболочку. Находясь в том же каталоге, где файл, вызовите из командной строки модуль `unittest`:

```
$ python -m unittest test_example.MyTest
.
----------------------------------------------------------------------
Ran 1 test in 0.000s
OK
```

Для запуска всех тестов из файла укажите файл:

```
$ python -m unittest test_example
.
----------------------------------------------------------------------
Ran 2 tests in 0.000s
OK
```

Mock (в модуле unittest)

В версии Python 3.3 `unittest.mock` (https://docs.python.org/dev/library/unittest.mock) доступен в стандартной библиотеке. Он позволяет заменять тестируемые части системы mock-объектами и делать предположения о том, как они используются.

Например, вы можете написать *обезьяний патч* для метода, похожий на тот, что показан в предыдущем примере (обезьяний патч — это код, который модифицирует или заменяет другой существующий код во время работы программы). В этом коде существующий метод с именем `ProductionClass.method` (в случае если мы создали именованный объект) заменяется новым объектом `MagicMock`, который при вызове всегда будет возвращать значение `3`. Кроме того, этот объект считает количество получаемых вызовов, записывает сигнатуру, с помощью которой был вызван, и содержит методы с выражением, необходимые для тестов:

```
from unittest.mock import MagicMock
instance = ProductionClass()
instance.method = MagicMock(return_value=3)
instance.method(3, 4, 5, key='value')
instance.method.assert_called_with(3, 4, 5, key='value')
```

Для того чтобы создавать mock-классы и объекты при тестировании, используйте декоратор `patch`. В следующем примере поиск во внешней системе заменяется mock-объектом, который всегда возвращает одинаковый результат (патч существует только во время работы теста):

```
import unittest.mock as mock
def mock_search(self):
    class MockSearchQuerySet(SearchQuerySet):
        def __iter__(self):
            return iter(["foo", "bar", "baz"])
    return MockSearchQuerySet()
# SearchForm относится к ссылке на импортированный класс
# myapp.SearchForm и модифицирует этот объект, но не код,
# где определяется сам класс SearchForm
```

```
@mock.patch('myapp.SearchForm.search', mock_search)
def test_new_watchlist_activities(self):
    # get_search_results выполняет поиск и итерирует по результату
    self.assertEqual(len(myapp.get_search_results(q="fish")), 3)
```

Вы можете сконфигурировать модуль `mock` и управлять его поведением разными способами. Они подробно описаны в документации к `unittest.mock`.

doctest

Модуль `doctest` выполняет поиск фрагментов текста, которые похожи на интерактивные сессии Python в строках документации, а затем выполняет эти сессии, чтобы убедиться, что они работают именно так, как было показано.

Модуль `doctest` служит другой цели, нежели юнит-тесты. Они обычно менее детальны и не отлавливают особые случаи или регрессионные ошибки. Вместо этого они выступают в качестве содержательной документации основных вариантов использования модуля и его компонентов (в качестве примера можно рассмотреть сценарий «счастливый путь» (happy path — https://en.wikipedia.org/wiki/Happy_path)). Однако такие тесты должны запускаться автоматически каждый раз, когда запускается весь набор тестов.

Рассмотрим простой пример `doctest`:

```
def square(x):
    """Squares x.
    >>> square(2)
    4
    >>> square(-2)
    4
    """
    return x * x if __name__ == '__main__':
    import doctest
    doctest.testmod()
```

Когда вы запускаете этот модуль из командной строки (например, с помощью команды `python module.py`), такие тесты начнут выполняться и «пожалуются», если какой-то компонент ведет себя не так, как описано в строках документации.

Примеры

В этом разделе мы рассмотрим фрагменты наших любимых пакетов для того, чтобы подчеркнуть правила хорошего тона при тестировании реального кода. Набор тестов предполагает наличие дополнительных библиотек, не включенных в эти пакеты (например, для Requests требуется Flask, чтобы создать mock-сервер HTTP), которые включены в файлы `requirements.txt` их проектов.

Для всех этих примеров ожидаемым первым шагом будет открытие терминальной оболочки, изменение каталогов таким образом, чтобы они указывали на то место, где лежат исходники к вашим проектам, а также клонирование репозитория исходного кода и настройка виртуальной среды. Например, так:

```
$ git clone https://github.com/username/projectname.git
$ cd projectname
$ virtualenv -p python3 venv
$ source venv/bin/activate
(venv)$ pip install -r requirements.txt
```

Пример: тестирование в Tablib

Tablib использует модуль `unittest` стандартной библиотеки Python. Набор тестов не поставляется с пакетом. Для получения файлов вы должны клонировать репозиторий GitHub. Приводим основные моменты, выделив главные части.

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
"""Тесты для Tablib."""

import json
import unittest
import sys
import os
import tablib
from tablib.compat import markup, unicode, is_py3
from tablib.core import Row

class TablibTestCase(unittest.TestCase):  (1)
    """Тестовый случай."""

    def setUp(self):  (2)
        """Создаем простой набор данных с заголовками."""

        global data, book

        data = tablib.Dataset()
        book = tablib.Databook()

        #
        #  ... пропускаем подготовительный код,
        #  который здесь не используется ...
        #

    def tearDown(self):  (3)
        """Очистка."""
        pass
```

```
    def test_empty_append(self):  ❹
        """Убедимся, что функция append() корректно добавляет
        кортеж без заголовков."""
        data.append(new_row)

        # Проверяем ширину/данные
        self.assertTrue(data.width == len(new_row))
        self.assertTrue(data[0] == new_row)

    def test_empty_append_with_headers(self):  ❺
        """Проверяем, что функция append() корректно определяет
        несовпадение длины заголовков и данных."""
        data.headers = ['first', 'second']
        new_row = (1, 2, 3, 4)

        self.assertRaises(tablib.InvalidDimensions, data.append,
        new_row)
```

❶ Для того чтобы использовать юнит-тест, создайте подкласс `unittest.TestCase` и напишите методы для тестирования, чьи имена начинаются с `test`. Класс `TestCase` предоставляет методы с выражением, которые позволяют выполнить проверку на равенство, правдивость, тип данных и наличие исключений (см. документацию по адресу http://bit.ly/unittest-testcase для получения более подробной информации).

❷ Метод `TestCase.setUp()` запускается всякий раз перед каждым методом `TestCase`.

❸ Метод `TestCase.tearDown()` запускается всякий раз после каждого метода `TestCase`[1].

❹ Все имена тестов должны начинаться со слова `test`, иначе они не запустятся.

❺ В одном тестовом случае может быть несколько тестов, но каждый из них должен тестировать что-то одно.

Если вы хотите внести вклад в Tablib, первое, что можете сделать после клонирования репозитория, — запустить набор тестов и убедиться, что все работает как полагается. Это можно сделать так:

---

[1] Обратите внимание, что метод unittest.TestCase.tearDown не будет запущен, если в коде есть ошибки. Это может удивить вас, если вы использовали функциональность unittest.mock для того, чтобы изменить реальное поведение кода. В Python 3.1 был добавлен метод unittest.TestCase.addCleanup(). Он помещает функцию очистки и ее аргументы в стек, и эта функция будет вызвана либо после метода unittest.TestCase.tearDown(), либо в любом случае независимо от того, был ли вызван метод tearDown(). Для получения более подробной информации обратитесь к документации метода unittest.TestCase.addCleanup() (см. http://docs.python.org/3/library/unittest.html#unittest.TestCase.addCleanup).

```
(venv)$ ### внутри каталога высшего уровня, tablib/
(venv)$ python -m unittest  test_tablib.py
.............................................................
----------------------------------------------------------------------
Ran 62 tests in 0.289s
OK
```

В версии Python 2.7 метод `unittest` также содержит собственный механизм обнаружения тестов, который доступен с помощью параметра `discover` в командной строке:

```
(venv)$ ### *above* the top-level directory, tablib/
(venv)$ python -m unittest discover tablib/
.............................................................
----------------------------------------------------------------------
Ran 62 tests in 0.234s
OK
```

После того как вы убедитесь, что все тесты проходят, вы: а) найдете тестовый случай, связанный с изменяемой частью проекта, и будете часто запускать его при изменении кода; б) напишете новый тестовый случай для функциональности, которую хотите добавить, или для ошибки, которую отслеживаете, и будете часто запускать его при изменении кода. Рассмотрим в качестве примера следующий сниппет:

```
(venv)$ ### внутри каталога высшего уровня, tablib/
(venv)$ python -m unittest test_tablib.TablibTestCase.test_empty_append
.
----------------------------------------------------------------------
Ran 1 test in 0.001s
OK
```

Как только ваш код начнет работать, снова задействуйте весь набор тестов перед тем, как отправить его в репозиторий. Поскольку вы часто запускаете тесты, они должны быть максимально быстрыми. Более подробную информацию о том, как использовать метод `unittest`, смотрите в документации по адресу http://bit.ly/unittest-library.

Пример: тестирование с помощью Requests

Пакет Requests использует `py.test`. Чтобы увидеть его в действии, откройте терминальную оболочку, перейдите во временный каталог, клонируйте Requests, установите все зависимости и запустите файл `py.test`, как показано здесь:

```
$ git clone -q https://github.com/kennethreitz/requests.git
$
$ virtualenv venv -q -p python3  # dash -q for 'quiet'
$ source venv/bin/activate
(venv)$
(venv)$ pip install -q -r requests/requirements.txt   # 'quiet' again...
(venv)$ cd requests
(venv)$ py.test
```

```
======================== test session starts ================================
platform darwin -- Python 3.4.3, pytest-2.8.1, py-1.4.30, pluggy-0.3.1
rootdir: /tmp/requests, inifile:
plugins: cov-2.1.0, httpbin-0.0.7
collected 219 items
tests/test_requests.py ....................................................
X.......................................
tests/test_utils.py ..s.....................................................
========= 217 passed, 1 skipped, 1 xpassed in 25.75 seconds ==================
```

Другие популярные инструменты

Инструменты для тестирования, перечисленные здесь, используются не так часто, но все еще достаточно популярны.

pytest

pytest (http://pytest.org/latest/) — это нешаблонная альтернатива модуля стандартной библиотеки Python. Это означает, что для него не требуется создавать временные платформы для тестовых случаев и, возможно, даже не нужны методы установки и очистки. Для установки запустите команду `pip` в обычном режиме:

```
$ pip install pytest
```

Несмотря на то что инструмент тестирования имеет множество возможностей и его можно расширять, синтаксис остается довольно простым. Создать набор тестов так же просто, как и написать модуль с несколькими функциями:

```
# содержимое файла test_sample.py
def func(x):
    return x + 1
def test_answer():
    assert func(3) == 5
```

После этого вам лишь нужно вызвать команду `py.test`. Сравните это с работой, которая потребуется для создания эквивалентной функциональности с помощью модуля `unittest`:

```
$ py.test
========================== test session starts ===========================
platform darwin -- Python 2.7.1 -- pytest-2.2.1
collecting ... collected 1 items
test_sample.py F
================================ FAILURES ================================
_____________________________ test_answer _____________________________
    def test_answer():
>       assert func(3) == 5
E       assert 4 == 5
E        +  where 4 = func(3)
test_sample.py:5: AssertionError
======================== 1 failed in 0.02 seconds ========================
```

Nose

Nose (http://readthedocs.org/docs/nose/en/latest/) расширяет `unittest` для того, чтобы упростить тестирование:

```
$ pip install nose
```

Предоставляет возможность автоматически обнаруживать тесты, чтобы сэкономить ваше время и избавить от необходимости создавать наборы тестов вручную. Предлагает множество надстроек для дополнительной функциональности вроде совместимого с xUnit вывода тестов, отчетов о покрытии, а также выбора тестов.

tox

tox (http://testrun.org/tox/latest/) — инструмент для автоматизирования управления средами тестирования и для тестирования в разных конфигурациях интерпретатора:

```
$ pip install tox
```

tox позволяет сконфигурировать сложные матрицы тестов с большим количеством параметров с помощью конфигурационного файла, похожего на INI-файлы.

Варианты для старых версий Python

Если вы не можете контролировать свою версию Python, но хотите использовать эти инструменты тестирования, предлагаем вам несколько вариантов.

unittest2. Это обратный порт модуля unittest (http://pypi.python.org/pypi/unittest2) для версии Python 2.7, который имеет усовершенствованный API и лучшие выражения относительно тех, что были доступны в предыдущих версиях Python.

Если вы используете Python 2.6 или ниже (например, если вы работаете в крупном банке или компании Fortune 500), можете установить его с помощью команды `pip`:

```
$ pip install unittest2
```

Вы можете захотеть импортировать модуль под именем `unittest`, чтобы вам было проще портировать код на новые версии модуля в будущем:

```
import unittest2 as unittest
class MyTest(unittest.TestCase):
    ...
```

Таким образом, если вы когда-нибудь перейдете на новую версию Python и вам больше не потребуется модуль `unittest2`, вы сможете изменить выражение импорта, не меняя остальной код.

Mock. Если вам понравилось то, что вы прочитали в пункте «Mock (в модуле unittest)» раздела «Основы тестирования» выше, но вы работаете с Python в версии ниже 3.3, вы все еще можете использовать `unittest.mock`, импортировав его как отдельную библиотеку:

```
$ pip install mock
```

fixture. Предоставляет инструменты, которые позволяют проще настраивать и очищать бэкенды баз данных для тестирования (http://farmdev.com/projects/fixture/). Он может загружать фальшивые наборы данных для использования в SQLAlchemy, SQLObject, Google Datastore, Django ORM и Storm. Существуют и его новые версии, но его тестировали только для версий Python 2.4-2.6.

Lettuce и Behave

Lettuce и Behave — это пакеты для выполнения *разработки через реализацию поведения* (behavior-driven development, BDD) в Python. BDD — процесс, который появился на основе TDD в начале 2000-х годов для того, чтобы заменить слово «тест» в TDD на слово «поведение» (дабы преодолеть проблемы, возникающие у новичков при освоении TDD). Это название появилось благодаря Дэну Норту (Dan North) в 2003-м и было представлено миру наряду с инструментом JBehave для Java в статье 2006 года в журнале *Better Socware* (представляла собой отредактированную статью из блога Дэна Норта *Introducing BDD* по адресу http://dannorth.net/introducing-bdd).

Концепция BDD набрала популярность после того, как в 2011 году вышла книга *6e Cucumber Book* (Pragmatic Bookshelf), где был задокументирован пакет Behave для Ruby. Это вдохновило Гэбриэла Фалько (Gabriel Falco) на создание Lettuce (http://lettuce.it/), а Питера Паренте (Peter Parente) — на создание Behave (http://pythonhosted.org/behave/) для нашего сообщества.

Поведения описываются простым текстом с помощью синтаксиса под названием Gherkin, который люди могут прочитать, а машины — понять. Вам могут пригодиться следующие руководства:

- руководство по Gherkin (https://github.com/cucumber/cucumber/wiki/Gherkin);
- руководство по Lettuce (http://lettuce.it/tutorial/simple.html);
- руководство по Behave (http://tott-meetup.readthedocs.org/en/latest/sessions/behave.html).

Документация

Читаемость — главная цель разработчиков Python как в проектах, так и в документации. Приемы, описанные в этом разделе, помогут вам сэкономить немало времени.

Документация к проекту

Существует документация по API, предназначенная пользователям проектов, а также дополнительная документация для тех, кто хочет вносить в проект свой вклад. В этом разделе вы узнаете о дополнительной документации.

Файл `README`, расположенный в корневом каталоге, призван давать общую информацию как пользователям, так и тем, кто обслуживает проект. В нем должен быть либо простой текст, либо легкая для чтения разметка вроде reStructured Text (сейчас это единственный формат, который понимает PyPI[1]) или Markdown (https://help.github.com/articles/basic-writing-and-formatting-syntax/). Этот файл должен содержать несколько строк, описывающих предназначение проекта или библиотеки (предполагая, что пользователь ничего не знает о проекте), URL основного исходного кода ПО и информацию об авторах. Если вы планируете читать код, то в первую очередь должны ознакомиться с этим файлом.

Файл `INSTALL` не особенно нужен в Python (но он может пригодиться для того, чтобы соответствовать требованиям лицензий вроде GPL). Инструкции по установке зачастую сокращаются до одной команды вроде `pip install module` или `python setup.py install` и добавляются в файл `README`.

Файл `LICENSE` должен присутствовать всегда и указывать лицензию, под которой ПО доступно общественности (см. раздел «Выбираем лицензию» далее в этой главе для получения более подробной информации.)

В файле `TODO` или одноименном разделе файла `README` должны быть представлены планы по развитию кода.

В файле `CHANGELOG` или одноименном разделе файла `README` должны быть приведены изменения, которые произошли с базой кода в последних версиях.

Публикация проекта

В зависимости от проекта ваша документация может содержать некоторые (или даже все) из этих компонентов:

- во *введении* должен даваться краткий обзор того, что можно сделать с продуктом (плюс один или два простых варианта использования). Этот раздел представляет собой 30-секундную речь, описывающую ваш проект;
- в *разделе «Руководство»* основные варианты использования описаны более подробно. Читатель пройдет пошаговую процедуру настройки рабочего прототипа;

[1] Для тех, кому интересно: развернулась дискуссия о введении поддержки (https://bitbucket.org/pypa/pypi/issues/148/support-markdown-for-readmes) в файлах README в PyPI.

- *раздел API* генерируется на основе кода (см. подраздел «Строки документации против блоковых комментариев» текущего раздела далее). В нем перечислены все доступные интерфейсы, параметры и возвращаемые значения;
- *документация для разработчиков* предназначена для тех, кто хочет внести свой вклад в проект. В ней могут содержаться соглашения, принятые в коде, и описываться общая стратегия проектирования.

Sphinx

Sphinx (http://sphinx.pocoo.org/) — самый популярный[1] инструмент для создания документации для Python. Используйте его: он преобразует язык разметки reStructured Text в огромное множество форматов, включая HTML, LaTeX (для печатаемых версий PDF), страницы руководства и простой текст.

Существует *отличный бесплатный* хостинг для вашей документации, созданной с помощью Sphinx: Read the Docs (http://readthedocs.org/). Используйте и его. Вы можете сконфигурировать его с помощью функций перехвата коммитов для вашего репозитория исходного кода, поэтому перестроение вашей документации будет происходить автоматически.

Sphinx знаменит благодаря генерации API. Он также хорошо работает для общей документации в проекте. Онлайн-версия книги «Автостопом по Python» создана с помощью Sphinx и размещена на сайте Read the Docs.

reStructured Text

Sphinx использует формат reStructured Text (http://docutils.sourceforge.net/rst.html), с его помощью написана практически вся документация для Python. Если содержимое аргумента `long_description` функции `setuptools.setup()` написано в формате reStructured Text, оно будет отрисовано как HTML в PyPI — другие форматы будут представлены как простой текст. Он похож на Markdown, имеющий встроенные необязательные расширения. Следующие ресурсы подойдут для изучения синтаксиса:

- The reStructuredText Primer (http://sphinx.pocoo.org/rest.html);
- reStructuredText Quick Reference (http://bit.ly/restructured-text).

Или начните вносить свой вклад в документацию к любимому проекту и учитесь в процессе чтения.

[1] Вы можете встретить и другие инструменты вроде Pycco, Ronn, Epydoc (больше не поддерживается) и MkDocs. Практически все используют Sphinx, мы также рекомендуем выбрать его.

Строки документации против блоковых комментариев

Строки документации и блоки комментариев не взаимозаменяемы. Оба варианта могут применяться для функции или класса. Рассмотрим пример использования обоих.

```
# Эта функция по какой-то причине замедляет выполнение программы. (1)
def square_and_rooter(x):
    """Возвращаем квадратный корень значения переменной, (2)
    умноженного на само себя."""
    ...
```

(1) Первый блок комментария — это заметка для программиста.

(2) Строки документации описывают, как работает функция или класс, она будет показана в интерактивной сессии Python, когда пользователь введет команду `help(square_and_rooter)`.

Строки документации, размещенные в начале модуля или в начале файла `__init__.py`, также появятся в выводе функции `help()`. Функция `autodoc` для Sphinx может автоматически генерировать документацию с помощью правильно отформатированных строк. Инструкцию о том, как форматировать строки документации для `autodoc`, вы можете прочитать в руководстве к Sphinx (http://www.sphinx-doc.org/en/stable/tutorial.html#autodoc). Для получения более подробных сведений обратитесь к PEP 257 (https://www.python.org/dev/peps/pep-0257/).

Журналирование

Модуль журналирования был частью стандартной библиотеки Python, начиная с версии 2.3. Он кратко описан в PEP 282 (https://www.python.org/dev/peps/pep-0282/). Эту документацию сложно прочесть, исключение составляет простое руководство по журналированию (http://docs.python.org/howto/logging.html#logging-basic-tutorial).

Журналирование бывает двух видов:

- *диагностическое журналирование* — записываются все события, связанные с работой приложения. Если пользователь сообщает об ошибке, в этих журналах можно поискать контекст;
- *ведение контрольных журналов* — записываются события для бизнес-анализа. Транзакции пользователя (вроде истории посещений) могут быть извлечены и объединены с другой информацией о нем (вроде итоговых покупок) для отчетности или оптимизации бизнес-целей.

Журналирование против функции print

Единственный случай, когда `print` предпочтительнее журналирования, — если вам нужно отобразить справку для приложения командной строки. Рассмотрим причины, почему журналирование лучше, чем `print`:

- запись в журнале (https://docs.python.org/library/logging.html#logrecord-attributes), которая создается при каждом событии журналирования, содержит полезную диагностическую информацию вроде имени файла, полного пути, функции и номера строки для события журналирования;
- к событиям, записанным во включенных модулях, вы можете получить доступ автоматически с помощью корневого средства ведения журнала в потоке журналирования для вашего приложения, если только вы их не отфильтруете;
- процесс журналирования можно выборочно приостанавливать с помощью метода `logging.Logger.setLevel()` или отключать путем установки значения атрибута `logging.Logger.disabled` равным `True`.

Журналирование для библиотеки

Заметки о конфигурировании журналирования для библиотеки содержатся в руководстве по журналированию (http://bit.ly/configuring-logging). Еще один хороший ресурс с примерами использования журналирования — библиотеки, которые мы упомянем в следующей главе. Поскольку пользователь (а не библиотека) должен указывать, что случится, когда произойдет событие журналирования, мы должны констатировать:

> Настоятельно рекомендуется не добавлять никаких обработчиков помимо NullHandler к средствам ведения журнала для библиотек.

Обработчик `NullHandler` делает то, что указано в его имени, то есть ничего. В противном случае пользователь должен будет самостоятельно отключать журналирование, если оно ему не требуется.

Правилом хорошего тона считается создание объектов средств ведения журнала только для использования вместе с переменной `__name__ global`: модуль журналирования создает иерархию средств ведения журнала с помощью точечной нотации, поэтому использование конструкции `__name__` гарантирует отсутствие пересечений.

Рассмотрим пример применения этого приема в исходном коде библиотеки Requests (https://github.com/kennethreitz/requests) — разместите это в файле верхнего уровня `__init__.py` вашего проекта:

```
# Установить дескриптор журналирования по умолчанию для того, чтобы
# избежать появления предупреждений, которые гласят «Обработчик не найден».
import logging
try:  # Python 2.7+
    from logging import NullHandler
except ImportError:
    class NullHandler(logging.Handler):
        def emit(self, record):
            pass
logging.getLogger(__name__).addHandler(NullHandler())
```

Журналирование для приложения

Twelve-Factor App (http://12factor.net/) (авторитетный источник, где перечислены правила хорошего тона, применяемые при разработке приложений) содержит раздел, в котором рассказывается о подобных правилах журналирования (http://12factor.net/logs). В нем предложено рассматривать события журнала как поток событий, для отправки этого потока в стандартный поток вывода нужно использовать среду приложения.

Существует минимум три способа конфигурирования средств ведения журнала (табл. 4.4).

Таблица 4.4. Способы конфигурирования средств ведения журнала

Способ	Плюсы	Минусы
Использование файла в формате INI	Вы можете обновлять конфигурацию при запуске функции logging.config.listen(), которая будет слушать изменения в сокете	У вас будет не такой полный контроль (например, пользовательские фильтры или средства ведения журнала, созданные как подклассы), чем это возможно при конфигурировании средств ведения журнала в коде
Использование словаря или файла в формате JSON	В дополнение к обновлению во время работы вы также можете загружать конфигурацию из файла с помощью модуля json, который находится в стандартной библиотеке, начиная с Python 2.6	У вас будет не такой полный контроль, чем это возможно при конфигурировании средств ведения журнала в коде
Использование кода	Вы имеете полный контроль над конфигурированием	Любые модификации потребуют внесения изменений в исходный код

Пример конфигурации с помощью файла в формате INI

Более подробная информация о формате INI содержится в разделе руководства журналирования, посвященном журналированию конфигурации (https://docs.py-

thon.org/howto/logging.html#configuring-logging). Минимальный файл конфигурации будет выглядеть так:

```
[loggers]
keys=root

[handlers]
keys=stream_handler

[formatters]
keys=formatter

[logger_root]
level=DEBUG
handlers=stream_handler

[handler_stream_handler]
class=StreamHandler
level=DEBUG
formatter=formatter
args=(sys.stderr,)

[formatter_formatter]
format=%(asctime)s %(name)-12s %(levelname)-8s %(message)s
```

`asctime`, `name`, `levelname` и `message` являются необязательными атрибутами библиотеки журналирования. Полный список доступных вариантов и их описание смотрите в документации Pytho (http://bit.ly/logrecord-attributes). Предположим, что наша конфигурация журналирования называется `logging_conf.ini`. Для того чтобы настроить средства ведения журнала с помощью этой конфигурации в коде, используем функцию `logging.config.fileconfig()`:

```
import logging
from logging.config import fileConfig

fileConfig('logging_config.ini')
logger = logging.getLogger()
logger.debug('often makes a very good meal of %s', 'visiting tourists')
```

Пример конфигурирования с помощью словаря

В версии Python 2.7 вы можете использовать словарь с деталями конфигурации. В PEP 391 (https://www.python.org/dev/peps/pep-0391) содержится список обязательных и необязательных элементов словаря конфигурации. Рассмотрим минимальную реализацию:

```
import logging
from logging.config dictConfig

logging_config = dict(
```

```
    version = 1,
    formatters = {
        'f' : {'format' :
                '%(asctime)s %(name)-12s %(levelname)-8s %(message)s' }
        },
    handlers = {
        'h': {'class': 'logging.StreamHandler',
                'formatter': 'f',
                'level': logging.DEBUG}
    loggers = {
        'root': {'handlers': ['h'],
                'level': logging.DEBUG}
        }
)
dictConfig(logging_config)

logger = debugging.getLogger()
logger.debug('often makes a very good meal of %s', 'visiting tourists')
```

Пример конфигурирования непосредственно в коде

Наконец, рассмотрим минимальную конфигурацию журналирования, расположенную непосредственно в коде:

```
import logging

logger = logging.getLogger()
handler = logging.StreamHandler()
formatter = logging.Formatter(
        '%(asctime)s %(name)-12s %(levelname)-8s %(message)s')
handler.setFormatter(formatter)
logger.addHandler(handler)
logger.setLevel(logging.DEBUG)

logger.debug('often makes a very good meal of %s', 'visiting tourists')
```

Выбираем лицензию

В Соединенных Штатах Америки, если для вашего исходного кода не указана лицензия, пользователи не получат законного права загружать, модифицировать или распространять его. Помимо этого, они не смогут вносить свой вклад в проект, если вы не укажете, по каким правилам играть. Поэтому вам *нужна* лицензия.

Лицензии

Если ваш проект основан на другом проекте, вам следует выбрать лицензию. Например, Python Software Foundation (PSF) просит всех, кто вносит свой вклад в исходный код, подписать соглашение для участников, которое формально лицензирует

их код под одной из двух лицензий PSF[1] (при этом они сохраняют авторские права). Поскольку обе лицензии позволяют сублицензировать код с другими условиями, PSF может свободно распространять Python под своей лицензией Python Software Foundation License. Вы можете ознакомиться с часто задаваемыми вопросами о лицензии PSF (https://wiki.python.org/moin/PythonSoftwareFoundationLicenseFaq) (простым языком описывается, что пользователи могут и не могут делать). При этом дальнейшее применение дистрибутивов Python от PSF за пределами лицензии не предполагается.

Доступные варианты

Существует множество лицензий. PSF рекомендует использовать лицензию, одобренную Open Source Institute (OSI) (http://opensource.org/licenses). Если вы хотите вносить свой код в PSF, вам будет проще это делать, начав работу с одной из лицензий, указанных на странице https://www.python.org/psf/contrib/.

Помните, что необходимо изменять текст заполнителя в шаблонах лицензий для того, чтобы добавить актуальную информацию. Например, шаблон для лицензии MIT содержит текст Copyright (c) <year> <copyright holders> во второй строке. Шаблон для лицензии Apache License, Version 2.0 не требует модификации.

Лицензии для открытого исходного кода, как правило, делятся на две категории[2].

- *Разрешительные*. Часто называются похожими на Berkeley Software Distribution (BSD), больше концентрируются на том, чтобы дать пользователю свободу относительно того, что он хочет сделать со своим ПО.

  Примеры:

  - лицензия Apache 2.0 (https://opensource.org/licenses/Apache-2.0) — это действующая лицензия, измененная таким образом, что пользователи могут включать ее, не изменяя проект, добавив ссылку на нее в каждый файл. Можно использовать код под лицензией Apache 2.0 с помощью GNU General Public License версии 3.0 (GPLv3);
  - лицензии BSD 2-clause (лицензия из двух пунктов) и BSD 3-clause (лицензия из трех пунктов) (https://opensource.org/licenses/BSD-3-Clause) — последняя представляет собой лицензию из двух пунктов, которая также содержит дополнительное ограничение использования торговых марок издателя;

[1] На момент написания книги они находились под лицензией Academic Free License v. 2.1 или Apache License, Version 2.0. Полное описание того, как это работает, вы можете найти на странице https://www.python.org/psf/contrib/.

[2] Все лицензии, описанные здесь, одобрены OSI, вы можете узнать о них на главной странице OSI по адресу https://opensource.org/licenses.

- лицензии Massachusetts Institute of Technology (MIT) (https://opensource.org/licenses/MIT) — версии Expat и X11 названы в честь популярных продуктов, использующих соответствующие лицензии;
- лицензия Internet Software Consortium (ISC) (https://opensource.org/licenses/ISC) практически идентична лицензии MIT, за исключением нескольких строк, сейчас она считается устаревшей.

- *Свободные*. Больше концентрируются на том, чтобы гарантировать, что исходный код, включая изменения, которые в него вносятся, будет доступен. Среди таких лицензий наиболее известно семейство GPL. Текущая версия лицензии этого семейства — GPLv3 (https://opensource.org/licenses/GPL-3.0).

Лицензия GPLv2 несовместима с Apache 2.0, поэтому код под лицензией GPLv2 нельзя объединять с кодом под лицензией Apache 2.0. Но проекты под этой лицензией могут быть использованы в проектах под лицензией GPLv3 (которые впоследствии тоже перейдут под лицензию GPLv3).

Лицензии, соответствующие критериям OSI, позволяют использовать код в коммерческих целях, модифицировать ПО и распространять его с разными ограничениями и требованиями. Все лицензии, перечисленные в табл. 4.5, ограничивают ответственность пользователя и требуют от него помнить об авторских правах и лицензии при любом распространении.

Таблица 4.5. Темы, рассматриваемые в популярных лицензиях

Семейство лицензий	Ограничения	Разрешения	Требования
BSD	Защитить торговую марку издателя (BSD 3-clause)	Дает гарантию (BSD 2-clause и BSD 3-clause)	—
MIT (X11 или Expat), ISC	Защитить торговую марку издателя (ISC и MIT/X11)	Разрешает сублицензирование под другой лицензией	—
Apache версии 2.0	Защитить торговую марку издателя	Разрешает сублицензирование, использование в патентах	Необходимо указывать изменения, вносимые в исходный код
GPL	Запрещает сублицензирование под другой лицензией	Дает гарантию и можно (только в GPLv3) использовать в патентах	Необходимо указывать изменения, вносимые в исходный код, и включать исходный код

Лицензирование ресурсов

Книга Вана Линдберга (Van Lindberg) *Intellectual Property and Open Source* (издательство O'Reilly) — отличный ресурс, посвященный юридическим вопросам в отношении ПО с открытым исходным кодом. Эта книга поможет вам изучить лицензии и юридические тонкости, связанные с интеллектуальной собственностью (торговые марки, патенты, авторские права), а также их влияние на программы с открытым исходным кодом. Если вас не особо волнуют юридические моменты и вы хотите что-то быстро выбрать, вам могут помочь следующие сайты:

- GitHub предоставляет удобное руководство (http://choosealicense.com/), где сравниваются все лицензии в рамках нескольких предложений;
- на ресурсе TLDRLegal[1] (http://tldrlegal.com/) перечислено, что можно и чего нельзя делать под каждой лицензией;
- список лицензий, одобренных OSI (http://opensource.org/licenses), содержит полный текст всех лицензий, прошедших проверку на соответствие Open Source Definition (что позволит свободно использовать, модифицировать и распространять ПО).

[1] Аббревиатура tl;dr означает «Too long; didn't read» («Слишком длинно, не читал») и, скорее всего, существовала как сокращение для редакторов, пока не стала популярной в Интернете.

5 Читаем отличный код

Программисты читают много кода. И один из основных принципов, лежащих в основе дизайна Python, — читаемость. Ключ к тому, чтобы стать хорошим программистом, — читать и понимать отличный код. Такой код обычно следует принципам, которые мы перечисляли в разделе «Стиль кода» в начале главы 4, и его предназначение легко понять.

В этой главе приводятся выдержки из наиболее читаемых проектов Python, которые иллюстрируют темы, рассмотренные в главе 4. По мере их описания мы также поделимся приемами чтения кода[1].

Перед вами список проектов, которые продемонстрированы в этой главе, они приведены в том порядке, в котором появляются:

- *HowDoI* (https://github.com/gleitz/howdoi) — консольное приложение, которое ищет в Интернете ответы на вопросы, связанные с программированием; написано на Python;
- *Diamond* (https://github.com/python-diamond/Diamond) — демон Python[2], который собирает метрики и публикует их на Graphite или других бэкендах. Может собирать метрики для процессора, памяти, сети, ввода-вывода, загрузки и дисков. Предоставляет API для реализации пользовательских сборщиков метрик из практически любого источника;
- *Tabli* (https://github.com/kennethreitz/tablib) — независимая от формата библиотека, позволяющая работать с таблицами данных;

---

[1] Если вам нужна книга, материал которой основан на десятилетнем опыте чтения и рефакторинга кода, рекомендуем вам Object-Oriented Reengineering Patterns (издательство Square Bracket Associates) (http://scg.unibe.ch/download/oorp/index.html) за авторством Сержа Демейера (Serge Demeyer), Стефана Дюкасса (Stéphane Ducasse) и Оскара Нирстраса (Oscar Nierstrasz).

[2] Демон — это компьютерная программа, которая работает как фоновый процесс.

- *Requests* (https://github.com/kennethreitz/requests) — библиотека для протокола передачи гипертекста (HyperText Transfer Protocol, HTTP) для людей (90 % из нас хотят иметь HTTP-клиент, который автоматически выполняет авторизацию и соответствует многим стандартам (https://www.w3.org/Protocols/) для выполнения таких заданий, как многокомпонентная загрузка файла с помощью единственного вызова функции);
- *Werkzeug* (https://github.com/mitsuhiko/werkzeug) изначально был коллекцией различных утилит для приложений стандарта Web Service Gateway Interface (WSGI), а теперь стал одним из наиболее мощных вспомогательных модулей WSGI;
- *Flask* (https://github.com/mitsuhiko/flask) — микрофреймворк для Python, основанный на Werkzeug и Jinja2. Подойдет для быстрого создания веб-страниц.

Эти проекты могут решать и другие задачи (не только те, что мы упомянули), и мы надеемся, что после прочтения текущей главы вы захотите загрузить и прочесть подробнее хотя бы один-два этих проекта самостоятельно (а затем, возможно, даже рассказать о них другим).

Типичные функции

Некоторые функции одинаковы у всех проектов: детали снепшотов для каждого из них показывают, что их функции состоят из очень малого количества строк кода (меньше 20, исключая пробелы и комментарии) и множества пустых строк. Крупные и более сложные проекты включают строки документации и/или комментарии; обычно больше пятой части содержимого базы кода составляет документация. Как вы можете видеть на примере HowDoI, в котором нет строк документации (поскольку он не предназначен для интерактивного использования), комментарии необязательны, если код простой и ясный. В табл. 5.1 описаны стандартные характеристики этих проектов.

Таблица 5.1. Типичные функции рассматриваемых проектов

Пакет	Лицензия	Количество строк	Строки документации (% от общего количества строк)	Комментарии документации (% от общего количества строк)	Пустые строки документации (% от общего количества строк)	Средняя длина функций
HowDoI	MIT	262	0	6	20	13 строк
Diamond	MIT	6021	21	9	16	11 строк
Tablib	MIT	1802	19	4	27	8 строк
Requests	Apache 2.0	4072	23	8	19	10 строк
Flask	BSD 3-clause	10 163	7	12	11	13 строк
Werkzeug	BSD 3-clause	25 822	25	3	13	9 строк

В каждом разделе мы будем использовать разные приемы чтения кода для того, чтобы понять, чему посвящен проект. Далее мы приведем фрагменты кода, которые демонстрируют темы, описанные в этом руководстве. (Если мы не выделили какую-то функцию в одном проекте, это не означает, что ее там нет; мы лишь хотим охватить большое количество концепций с помощью этих примеров.) После завершения чтения этой главы вы будете более уверенно работать с кодом. Приведенные примеры продемонстрируют, что такое хороший код (некоторые идеи вы, возможно, захотите применить в своем коде в будущем).

HowDoI

Проект HowDoI, написанный Бенджамином Гляйтсманом (Benjamin Gleitzman), станет отличной стартовой точкой нашей одиссеи, несмотря на то что он состоит менее чем из 300 строк.

Читаем сценарий, состоящий из одного файла

Сценарий обычно имеет четко определенные точку входа, параметры и точку выхода. Благодаря этому читать его проще, чем библиотеки, которые предоставляют API или фреймворк.

Загрузите модуль HowDoI с GitHub[1]:

```
$ git clone https://github.com/gleitz/howdoi.git
$ virtualenv -p python3 venv  # Или используйте mkvirtualenv, на ваш выбор...
$ source venv/bin/activate
(venv)$ cd howdoi/
(venv)$ pip install --editable .
(venv)$ python test_howdoi.py  # Запустите юнит-тесты
```

Теперь у вас должен быть установлен исполняемый файл `howdoi` в каталоге `venv/bin`. (Вы можете увидеть его, введя `cat 'which howdoi'` в командной строке.) Он был сгенерирован автоматически, когда вы ввели команду `pip install`.

Читаем документацию к HowDoI

Документация к HowDoI находится в файле `README.rst`, который располагается в репозитории на GitHub (https://github.com/gleitz/howdoi). Это небольшое приложение для командной строки, позволяющее пользователям искать в Интернете ответы на вопросы, связанные с программированием.

---

[1] Если вы столкнетесь с проблемой, когда lxml потребует более свежую версию библиотеки libxml2, установите более раннюю версию lxml, введя команду pip uninstall lxml; pip install lxml==3.5.0. Все сработает как надо.

В командной строке терминальной оболочки можно ввести команду `howdoi --help`, чтобы узнать, как пользоваться HowDoI:

```
(venv)$ howdoi --help
usage: howdoi [-h] [-p POS] [-a] [-l] [-c] [-n NUM_ANSWERS] [-C] [-v]
              [QUERY [QUERY ...]]
instant coding answers via the command line
positional arguments:
  QUERY                 the question to answer
optional arguments:
  -h, --help            show this help message and exit
  -p POS, --pos POS     select answer in specified position (default: 1)
  -a, --all             display the full text of the answer
  -l, --link            display only the answer link
  -c, --color           enable colorized output
  -n NUM_ANSWERS, --num-answers NUM_ANSWERS
                        number of answers to return
  -C, --clear-cache     clear the cache
  -v, --version         displays the current version of howdoi
```

Из документации мы знаем, что HowDoI получает ответы на вопросы, связанные с программированием, из Интернета. В руководстве указано, что можно выбрать ответ в определенной позиции, раскрасить выводимую информацию, получить несколько ответов, а также то, что модуль имеет кэш, который можно очистить.

Использование HowDoI

Мы можем подтвердить, что понимаем, как работает HowDoI. Рассмотрим пример:

```
(venv)$ howdoi --num-answers 2 python lambda function list comprehension
--- Answer 1 ---
[(lambda x: x*x)(x) for x in range(10)]
--- Answer 2 ---
[x() for x in [lambda m=m: m for m in [1,2,3]]]
# [1, 2, 3]
```

Мы установили HowDoI, прочли его документацию и теперь можем его использовать. Перейдем к чтению кода!

Читаем код HowDoI

Если вы заглянете в каталог `howdoi/`, то увидите два файла: `__init__.py`, который состоит всего из одной строки, указывающей номер версии, и `howdoi.py`, который мы откроем и прочитаем.

Просматривая файл `howdoi.py`, мы увидим, что каждое новое определение функции использовано в следующей функции; это упрощает чтение кода. Каждая функция выполняет всего одну задачу (она вынесена в ее имя). Главная функция `command_line_runner()` располагается в нижней части файла `howdoi.py`.

Вместо того чтобы приводить здесь исходный код HowDoI, мы можем проиллюстрировать структуру его вызовов с помощью графа, показанного на рис. 5.1. Этот граф создан с помощью Python Call Graph (https://pycallgraph.readthedocs.io/), который предоставляет визуализацию функций, вызываемых при запуске сценария Python. Он хорошо работает с приложениями командной строки благодаря тому, что они имеют одну точку входа и относительно небольшое количество путей выполнения, по которым может пойти код. (Обратите внимание, что мы вручную удалили из отрисованного изображения функции, которые отсутствуют в проекте HowDoI, дабы вместить граф на страницу, а также немного переформатировали его.)

Код мог бы выглядеть как одна большая спагетти-функция, сложная для восприятия. Вместо этого код был намеренно разбит на отдельные функции, имеющие понятные имена. Кратко рассмотрим граф, изображенный на рис. 5.1: функция `command_line_runner()` анализирует входные данные и передает флаги и запрос, полученные от пользователя, в функцию `howdoi()`. Далее функция `howdoi()` оборачивает функцию `_get_instructions()` в блок `try/except`, чтобы можно было отловить ошибки соединения и вывести адекватное сообщение об ошибке (поскольку код приложения не должен завершать работу при наличии исключения).

Основную работу делает функция `_get_instructions()`: она вызывает функцию `_get_links()`, чтобы выполнить поиск ссылок, соответствующих запросу, в Google или на сайте Stack Overflow, а затем — функцию `_get_answer()` для каждого полученного результата (вплоть до предельного количества ссылок, указанного пользователем в командной строке, — по умолчанию одной).

Функция `_get_answer()` следует ссылке на ресурс Stack Overflow, извлекает код из ответа, раскрашивает его и возвращает функции `_get_instructions()`, которая объединяет все ответы в одну строку и возвращает их. Функции `_get_links()` и `_get_answer()` вызывают `_get_result()`, чтобы выполнить HTTP-запрос: `_get_links()` для запроса Google и `_get_answer()` для получения ссылок из запроса Google.

Функция `_get_result()` лишь оборачивает функцию `requests.get()` блоком `try/except`, чтобы можно было отловить ошибки SSL, вывести сообщение об ошибке и повторно сгенерировать исключение, дабы блок `try/except` верхнего уровня мог отловить его и выйти (это правило хорошего тона для любых программ).

Упаковка HowDoI

Файл `setup.py` проекта HowDoI, который находится выше каталога `howdoi/`, — отличный пример модуля установки, поскольку в дополнение к обычной установке пакета он также устанавливает исполняемый файл (к которому вы можете обратиться при упаковке собственной утилиты командной строки). Функция установки `tools.setup()` использует аргументы с ключевым словом для определения всех параметров конфигурации. Та часть, которая отвечает за исполняемый файл, использует аргумент с ключевым словом `entry_points`.

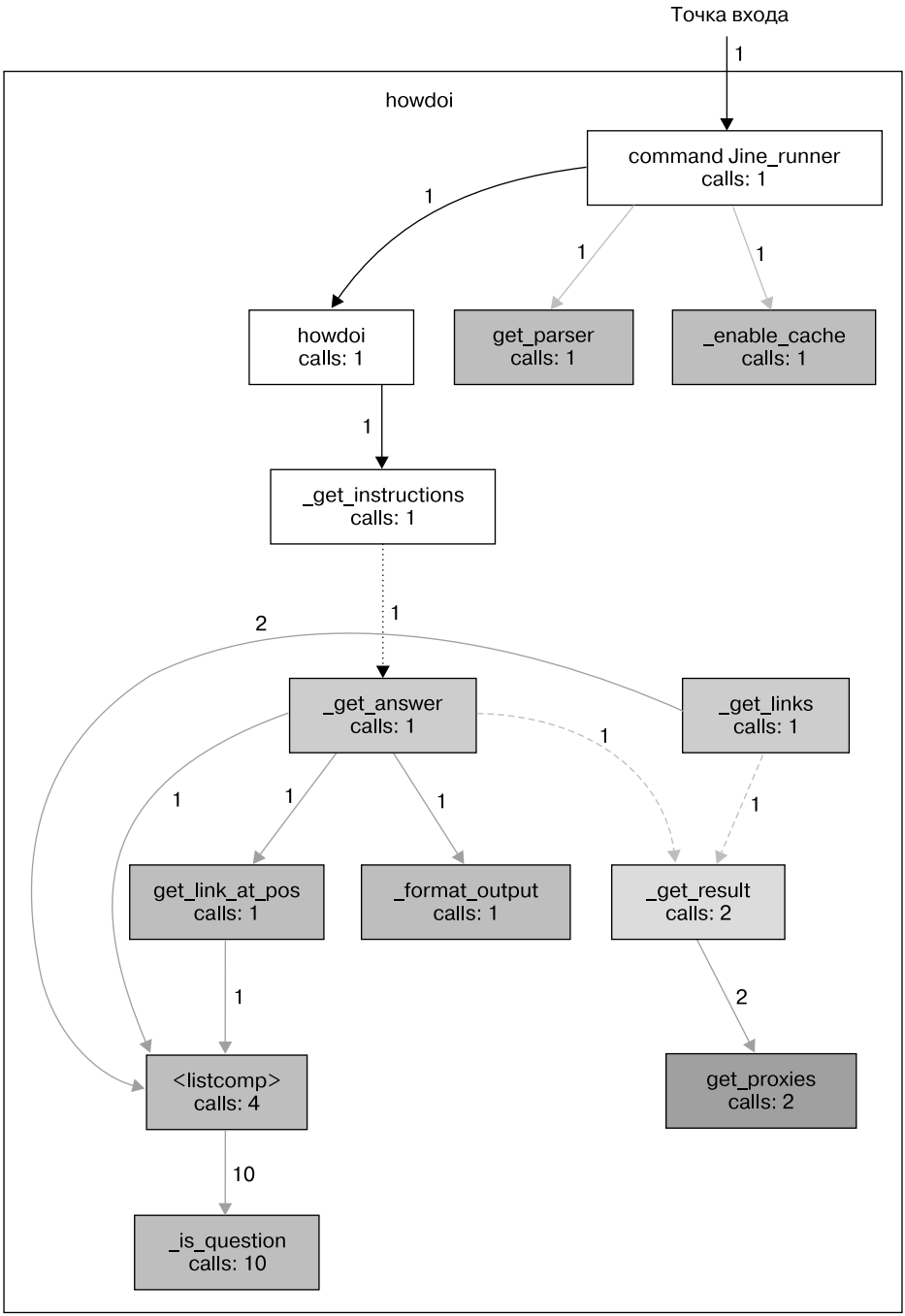

Рис. 5.1. Прозрачные пути и прозрачные имена функций в графе вызовов проекта howdoi

```
setup(
    name='howdoi',
    ##~~ ... Пропускаем другие типичные записи ...
    entry_points={
        'console_scripts': [  ❶
            'howdoi = howdoi.howdoi:command_line_runner',  ❷
        ]
    },
    ## ~~  ... Пропускаем остальные зависимости ...
)
```

❶ Ключевое слово для перечисления сценариев консоли — `console_scripts`.

❷ Объявляет, что исполняемый файл с именем `howdoi` в качестве цели будет иметь функцию `howdoi.howdoi.command_line_runner()`. Поэтому мы будем знать, что функция `command_line_runner()` является стартовой точкой для всего приложения.

Примеры из структуры HowDoI

HowDoI — это небольшая библиотека (в других разделах мы рассмотрим ее архитектуру более подробно, а здесь лишь скажем пару слов).

Пусть каждая функция делает что-то одно

Мы не устанем повторять, насколько полезно разделять внутренние функции HowDoI таким образом, чтобы они делали лишь что-то одно. Существуют функции, чье единственное предназначение — оборачивание других функций в оператор `try/except`. (Единственная функция, имеющая оператор `try/except`, которая не следует этой практике, — `_format_output()`. Она задействует операторы `try/except` не для обработки исключений, а для определения языка программирования с целью подсветки синтаксиса.)

Пользуйтесь данными, доступными системе

HowDoI проверяет и использует текущие системные значения, например с помощью функции `urllib.request.getproxies()` обрабатывается применение прокси-серверов (это подойдет для организаций вроде школ, которые имеют промежуточный сервер, фильтрующий соединение с Интернетом). Обратите внимание на этот сниппет:

```
XDG_CACHE_DIR = os.environ.get(
    'XDG_CACHE_HOME',
    os.path.join(os.path.expanduser('~'), '.cache')
)
```

Как вам узнать, что эти переменные существуют? Необходимость `urllib.request.getproxies()` обусловлена необязательными аргументами в `requests.get()`, поэтому часть этой информации доступна из сведений API о вызываемых вами библиотеках. Переменные среды зачастую нужны для определенной функциональности, поэтому, если библиотека предназначена для использования с конкретной базой данных или другим родственным приложением, в документации к этим приложениям будут перечислены актуальные переменные среды. Для систем POSIX хорошей стартовой точкой будет список стандартных переменных среды Ubuntu (https://help.ubuntu.com/community/EnvironmentVariables) или список переменных среды в спецификации POSIX (http://bit.ly/posix-env-variables), который указывает на другие важные списки.

Примеры из стиля HowDoI

HowDoI в основном следует PEP 8, но не в тех случаях, когда это вредит читаемости. Например, операторы импорта находятся в верхней части файла, но стандартная библиотека и внешние модули перемешаны. И несмотря на то что строковые константы в `USER_AGENTS` гораздо длиннее 80 символов, в строках нет места, где их можно было бы разбить естественным образом, поэтому они остаются целыми.

В нескольких следующих фрагментах кода показываются другие решения, связанные со стилем, о которых мы говорили в главе 4.

Имена функций, которые начинаются с нижнего подчеркивания (мы все — ответственные пользователи)

Имя почти каждой функции в HowDoI начинается с нижнего подчеркивания. Это показывает, что функции предназначены лишь для внутреннего использования. Для большинства из них это нужно, потому что при их прямом вызове появляется вероятность генерации необработанного исключения (этим грешат все функции, которые вызывают функцию `_get_result()`), до тех пор пока не будет вызвана функция `howdoi()`, обрабатывающая все возможные исключения.

Остальные внутренние функции (`_format_output()`, `_is_question()`, `_enable_cache()` и `_clear_cache()`) не предназначены для использования за пределами пакета. Тестирующий сценарий `howdoi/test_howdoi.py` вызывает только функции без префиксов, проверяя, что средство форматирования работает, и передавая аргумент командной строки для раскрашивания в функцию верхнего уровня `howdoi.howdoi()` вместо того, чтобы передавать код в функцию `howdoi._format_output()`.

Обрабатывайте вопросы совместимости только в одном месте (читаемость имеет значение)

Разница между версиями возможных зависимостей обрабатывается перед выполнением основного кода, поэтому пользователь знает, что проблем с зависимостями не возникнет, а проверка версий не засоряет код в других местах. Это хорошо, поскольку HowDoI поставляется как инструмент командной строки и дополнительные усилия, затрачиваемые пользователями, будут означать, что им не потребуется менять свою среду Python, чтобы установить инструмент. Рассмотрим фрагмент кода, в котором решается эта проблема:

```
try:
    from urllib.parse import quote as url_quote
except ImportError:
    from urllib import quote as url_quote
try:
    from urllib import getproxies
except ImportError:
    from urllib.request import getproxies
```

В следующем сниппете разница подходов к обработке Unicode в Python 2 и Python 3 нивелируется всего за семь строк путем создания функции `u(x)`, которая либо эмулирует Python 3, либо не делает ничего. Кроме того, он следует новым принципам цитирования Stack Overflow (http://meta.stackexchange.com/questions/271080), приводя оригинальный исходный код:

```
# Разрешаем разницу обработки Unicode между Python 2 и 3
# http://stackoverflow.com/a/6633040/305414
if sys.version < '3':
    import codecs
    def u(x):
        return codecs.unicode_escape_decode(x)[0]
else:
    def u(x):
        return x Питонские решения (красивое лучше, чем уродливое)
```

В следующем фрагменте кода из файла `howdoi.py` показываются продуманные, питонские решения. Функция `get_link_at_pos()` возвращает значение `False`, если результаты не найдены, или определяет ссылки на вопросы, касающиеся Stack Overflow, и возвращает ту из них, которая находится на желаемой позиции (либо последнюю, если ссылок недостаточно).

```
def _is_question(link):  (1)
    return re.search('questions/\d+/', link)

# [ ... пропускаем функцию ... ]

def get_link_at_pos(links, position):
    links = [link for link in links if _is_question(link)]  (2)
```

```
    if not links:
        return False  ❸

    if len(links) >= position:
        link = links[position-1]  ❹
    else:
        link = links[-1]  ❺
    return link  ❻
```

❶ Функция `_is_question()` определяется в отдельной строке, что указывает четкое значение непонятному в противном случае поиску с использованием регулярных выражений.

❷ Списковое включение читается как предложение благодаря отдельному определению функции `_is_question()` и информативным именам переменных.

❸ Раннее использование оператора возврата упрощает код.

❹ Дополнительный шаг, который тратится на присваивание значения переменной `link`, здесь...

❺ ...и здесь вместо использования двух отдельных операторов возврата, не имеющих именованных переменных, подчеркивает предназначение функции `get_link_at_pos()` с помощью прозрачных имен переменных. Код самозадокументирован.

❻ Единый оператор возврата, находящийся на высшем уровне отступов, явно показывает, что все пути по коду завершатся либо сразу (поскольку ссылки не найдены), либо в конце функции, вернув ссылку. Работает наше правило: мы можем прочесть первую и последнюю строки этой функции и понять, что она делает. (Получив несколько ссылок и позицию, функция `get_link_at_pos()` возвращает одну ссылку, которая находится на заданной позиции.)

Diamond

Diamond — это демон (приложение, постоянно работающее как фоновый процесс), который собирает метрики системы и публикует их в программах вроде MySQL, Graphite (http://graphite.readthedocs.org/) (платформа с открытым исходным кодом, созданная компанией Orbitz в 2008 году, которая сохраняет, получает и по возможности строит графики на основе временных рядов) и др. У нас есть возможность взглянуть на хорошую структуру пакетов, поскольку Diamond состоит из нескольких файлов и гораздо крупнее HowDoI.

Читаем более крупное приложение

Diamond также является приложением командной строки, поэтому, как и в случае с HowDoI, существуют четкая стартовая точка и четкие пути выполнения, однако теперь поддерживающий код находится в нескольких файлах.

Загрузите Diamond с GitHub (в документации говорится, что программа работает только в ОС CentOS или Ubuntu, но код, находящийся в ее файле `setup.py`, позволяет ей запускаться на всех платформах. Однако отдельные команды, которые стандартные сборщики используют для наблюдения за памятью, дисковым пространством и другими системными метриками, отсутствуют в Windows). На момент написания этой книги программа все еще использует Python 2.7:

```
$ git clone https://github.com/python-diamond/Diamond.git
$ virtualenv -p python2 venv  # Она все еще несовместима с Python 3...
$ source venv/bin/activate
(venv)$ cd Diamond/
(venv)$ pip install --editable .
(venv)$ pip install mock docker-py  # Эта зависимость нужна для тестирования.
(venv)$ pip install mock  # Эта зависимость также нужна для тестирования.
(venv)$ python test.py  # Запускаем юнит-тесты.
```

Как и в случае с библиотекой HowDoI, сценарий установки Diamond добавляет исполняемые файлы `diamond` и `diamond-setup` в каталог `venv/bin/`. В этот раз они не генерируются автоматически, а являются заранее написанными сценариями и лежат в каталоге `Diamond/bin/`. В документации говорится, что файл `diamond` запускает сервер, а `diamond-setup` — необязательный инструмент, который позволяет пользователям интерактивно изменять настройки сборщика в конфигурационном файле.

Существует множество дополнительных каталогов, пакет `diamond` находится внутри каталога `Diamond/src`. Мы взглянем на файлы из каталогов `Diamond/src` (в которых содержится основной код), `Diamond/bin` (хранится исполняемый файл `diamond`) и `Diamond/conf` (содержится пример конфигурационного файла). Остальные каталоги и файлы могут представлять интерес для тех, кто хочет распространять подобные приложения (в рамках этой книги мы их опустим).

Читаем документацию к Diamond

Для начала можно попытаться понять идею проекта, взглянув на онлайн-документацию (http://diamond.readthedocs.io/). Цель Diamond — упрощение сборки системных метрик для кластеров машин. Появилась в 2011 году благодаря компании BrightCove, Inc., на сегодняшний день в ее базу кода внесли вклад более 200 человек.

После описания истории и предназначения в документации говорится, как установить и запустить демон: вам нужно лишь модифицировать предложенный файл конфигурации (у нас он находится по адресу `conf/diamond.conf.example`), поместить в стандартное место (`/etc/diamond/diamond.conf`) или по пути, который вы укажете в командной строке, — и вы готовы приступить к работе. Кроме того, на вики-странице проекта Diamond (https://github.com/BrightcoveOS/Diamond/wiki/Configuration) вы можете найти полезный раздел о конфигурации.

Из командной строки мы можем вывести на экран руководство по использованию с помощью команды `diamond --help`:

```
(venv)$ diamond --help
Usage: diamond [options]
Options:
  -h, --help            show this help message and exit
  -c CONFIGFILE, --configfile=CONFIGFILE
                        config file
  -f, --foreground      run in foreground
  -l, --log-stdout      log to stdout
  -p PIDFILE, --pidfile=PIDFILE
                        pid file
  -r COLLECTOR, --run=COLLECTOR
                        run a given collector once and exit
  -v, --version         display the version and exit
  --skip-pidfile        Skip creating PID file
  -u USER, --user=USER  Change to specified unprivileged user
  -g GROUP, --group=GROUP
                        Change to specified unprivileged group
  --skip-change-user    Skip changing to an unprivileged user
  --skip-fork           Skip forking (damonizing) process
```

Из него мы узнаем, что демон использует файл конфигурации; по умолчанию работает в фоновом режиме; имеет возможность журналирования. Вы можете указать файл PID (process ID, «идентификатор процесса»), протестировать сборщики, можете изменить пользователя и группу процесса. По умолчанию он демонизирует (создаст копию) процесс[1].

Используем Diamond

Для того чтобы еще лучше понять Diamond, запустим его. Нам нужен модифицированный файл конфигурации, который мы можем поместить в созданный нами каталог `Diamond/tmp`. Находясь в нем, введите следующий код:

```
(venv)$ mkdir tmp
(venv)$ cp conf/diamond.conf.example tmp/diamond.conf
```

[1] Демонизируя процесс, вы создаете его копию, открепляете идентификатор сессии и копируете его снова, поэтому процесс будет полностью отключен от терминала, из которого вы его запускаете. Недемонизированные программы завершают работу при закрытии терминала — возможно, вы видели предупреждающее сообщение «Вы действительно хотите закрыть терминал? Это приведет к завершению запущенных процессов», за которым идет список запущенных в данный момент процессов. Демонизированный процесс продолжит работу после закрытия окна терминала. Он называется демоном в честь демона Максвелла (https://ru.wikipedia.org/wiki/Демон_Максвелла) (умного демона, а не гнусного).

Далее отредактируйте файл `tmp/diamond.conf`, чтобы он выглядел так:

```
### Настройки для сервера
[server]
# Обработчики для опубликованных метрик    ❶
handlers = diamond.handler.archive.ArchiveHandler
user =    ❷
group =
# Каталог, откуда будут загружаться модули сборщиков    ❸
collectors_path = src/collectors/

### Настройки обработчиков    ❹
[handlers]
[[default]]
[[ArchiveHandler]]
log_file = /dev/stdout

### Настройки сборщиков
[collectors]
[[default]]

# Интервал опроса по умолчанию (в секундах)
interval = 20

### Включенные по умолчанию коллекторы
[[CPUCollector]]
enabled = True

[[MemoryCollector]]
enabled = True
```

Из этого примера конфигурации видно следующее.

❶ Существует несколько обработчиков, каждый из которых мы можем выбрать по имени класса.

❷ Мы можем управлять пользователем и группой, под которыми запущен демон (пустое значение означает, что будут задействованы текущий пользователь и группа).

❸ Мы можем указать путь, который будет применен для поиска модулей сборщиков. Так Diamond узнает, где находятся пользовательские подклассы класса `Collector` (мы явно указываем это в файле конфигурации).

❹ Конфигурацию обработчиков мы храним отдельно.

Далее запустите Diamond, указав, что журнал будет сохраняться по адресу `/dev/stdout` (будет использована стандартная конфигурация форматирования), что приложение не должно работать в фоновом режиме, что нужно опустить запись в файл PID и использовать новые файлы конфигурации:

```
(venv)$ diamond -l -f --skip-pidfile --configfile=tmp/diamond.conf
```

Для того чтобы завершить процесс, нажимайте Ctrl+C до тех пор, пока снова не появится командная строка. Журнал показывает, что делают сборщики и обработчики: сборщики собирают разные метрики (вроде объема общей, свободной памяти и памяти подкачки от `MemoryCollector`), которые обработчики форматируют и отправляют в разные точки назначения вроде Graphite, MySQL (в нашем тестовом случае — как сообщения журнала в `/dev/stdout`).

Читаем код Diamond

Для чтения более крупных проектов лучше использовать IDE — с их помощью вы можете быстро обнаружить оригинальное определение функций и классов исходного кода (при наличии определения можете найти все места в проекте, где задействованы функция или класс). Для использования этой функциональности укажите интерпретатору Python вашей IDE применять одну из виртуальных сред[1].

Вместо того чтобы разбирать каждую функцию, как мы сделали это для HowDoI, изучим рис. 5.2 (показаны операторы импорта). Схема демонстрирует, какие модули Diamond импортируют другие модули. Подобные рисунки помогают понять более крупные проекты. Можно начать с исполняемого файла `diamond` в левом верхнем углу и следовать всем операторам импорта в проекте Diamond. Помимо исполняемого файла `diamond`, в каждом квадрате указывается файл (модуль) или папка (пакет) в каталоге `src/diamond`.

Хорошо организованные и удачно названные модули Diamond позволяют понять идею кода, просто взглянув на схему: модуль `diamond` получает версию из `util`, затем настраивает журналирование с помощью `utils.log` и запускает экземпляр сервера с помощью `server`. Сервер импортирует почти все вспомогательные модули, используя классы `utils.classes`, чтобы получить доступ к обработчикам в `handler` и сборщикам, `config` — для чтения файла конфигурации и получения настроек для сборщиков (дополнительные пути для сборщиков, определенных пользователем), `scheduler` и `signals` — для установки интервала опроса для сборщиков, чтобы подсчитать их метрики, а также для настройки обработчиков и указания им приступать к обработке очереди метрик, которые нужно отправить.

Схема не включает в себя вспомогательные модули `convertor.py` и `gmetric.py`, используемые определенными сборщиками, а также более 20 реализаций обработчиков, определенных в подпакете `handler`, и более 100 реализаций сборщиков, определенных в каталоге проекта `Diamond/src/collectors/` (которые находятся в другом месте, если процесс установки Diamond отличается от того, который мы выполнили при чтении, то есть использовали дистрибутивы пакетов PyPI или Linux вместо исходного кода). Они импортируются с помощью функции

[1] В PyCharm вы можете сделать это, выбрав пункты меню PyCharm ▸ Preferences ▸ Project:Diamond ▸ Project Interpreter, а затем указав пути к интерпретатору Python в текущей виртуальной среде.

`diamond.classes.load_dynamic_class()`, которая затем вызывает функцию `diamond.util.load_class_from_name()` для загрузки классов на основе имен, представленных в строках конфигурационного файла, поэтому операторы импорта могут не вызывать их явно.

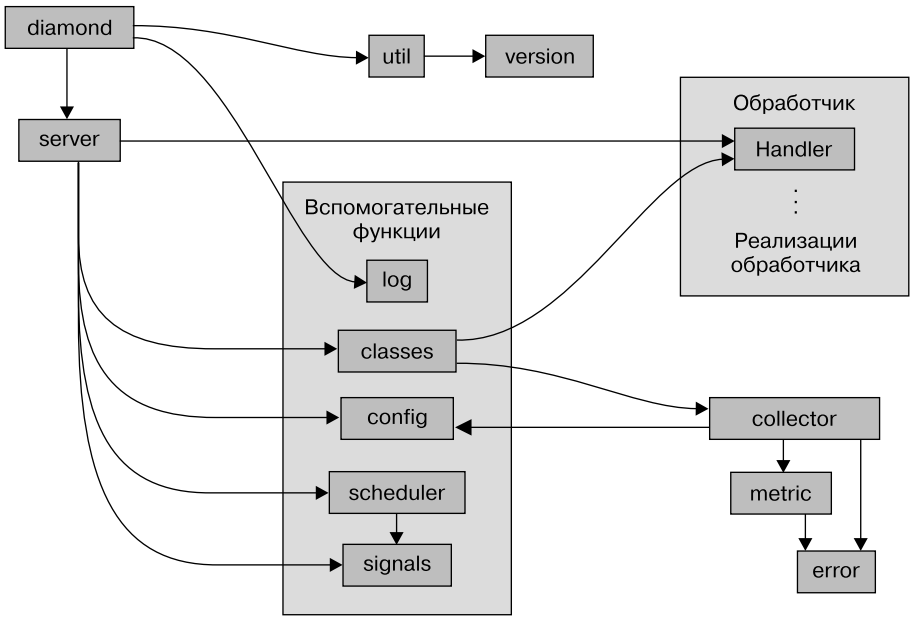

Рис. 5.2. Структура импортированных модулей в Diamond

Чтобы понять, для чего в проекте присутствуют пакет `utils` и модуль `util`, нужно открыть код: модуль `util` предоставляет функции, связанные с упаковкой Diamond (а не с его работой): функцию для получения номера версии на основе `version.__VERSION__` и две функции, которые анализируют строки, позволяющие определить модули или классы и импортировать их.

Журналирование в Diamond

Функция `diamond.utils.log.setup_logging()`, которая находится в файле `src/diamond/utils/log.py`, вызывается из функции `main()` исполняемого файла `diamond` при запуске демона:

```
# Инициализация журналирования
log = setup_logging(options.configfile, options.log_stdout)
```

Если значение `options.log_stdout` равно `True`, функция `setup_logging()` настроит средство ведения журнала со стандартным форматированием так, чтобы оно

отправляло записи в стандартный поток выхода на уровне DEBUG. Это делается в следующем фрагменте кода:

```
##~~ ... Пропускаем все остальное ...
def setup_logging(configfile, stdout=False):
    log = logging.getLogger('diamond')
    if stdout:
        log.setLevel(logging.DEBUG)
        streamHandler = logging.StreamHandler(sys.stdout)
        streamHandler.setFormatter(DebugFormatter())
        streamHandler.setLevel(logging.DEBUG)
        log.addHandler(streamHandler)
    else:
        ##~~ ... Пропускаем это ...
```

В противном случае он анализирует файл конфигурации с помощью функции `logging.config.file.fileConfig()` из стандартной библиотеки Python. Перед вами вызов функции — он выделен отступами, поскольку находится внутри предшествующего оператора `if/else` и блока `try/except`:

```
            logging.config.fileConfig(configfile,
                                      disable_existing_loggers=False)
```

Конфигурация журналирования игнорирует ключевые слова в конфигурационном файле, которые не связаны с журналированием. Так Diamond может использовать один и тот же конфигурационный файл как для своей конфигурации, так и для конфигурации журналирования. В примере конфигурационного файла, который располагается по адресу `Diamond/conf/diamond.conf.example`, среди прочих обработчиков определяется и обработчик журналирования:

```
### Настройки обработчиков
[handlers]
# обработчик(и) для журналирования
keys = rotated_file
```

Далее в файле конфигурации под заголовком «Настройки для журналирования» определяются примеры средств ведения журнала. Для получения более подробной информации смотрите документацию к конфигурационным файлам для журналирования (http://bit.ly/config-file-format).

Примеры из структуры Diamond

Diamond — это больше чем исполняемое приложение. Он также является библиотекой, которая предоставляет пользователям возможность создавать и применять собственные сборщики.

Мы продемонстрируем те элементы структуры пакета, которые нам нравятся, а затем изучим, как именно Diamond позволяет приложению импортировать и использовать определенные извне сборщики.

Разбиваем функциональность между пространствами имен (поскольку пространства имен — это отличная штука!)

На схеме рис. 5.2 показан модуль сервера, взаимодействующий с тремя другими модулями проекта: `diamond.handler`, `diamond.collector` и `diamond.utils`.

В подпакете `utils` можно было бы разместить все классы и функции в одном большом модуле `util.py`, однако можно подключить пространства имен для того, чтобы разбить код на отдельные файлы, — и команда разработчиков ею воспользовалась. Отличный выбор!

Все реализации обработчиков содержатся в каталоге `diamond/handler` (это логично), но структура для сборщиков отличается. Для них не предусмотрен каталог — только модуль `diamond/collector.py`, в котором определяются базовые классы `Collector` и `ProcessCollector`. Все реализации подклассов класса `Collector` определены в каталоге `Diamond/src/collectors/`, в виртуальной среде они будут установлены по адресу `venv/share/diamond/collectors`, если вы устанавливали Diamond с помощью PyPI (рекомендованный способ), а не с помощью GitHub (как это сделали мы). Это помогает пользователю создать новые реализации сборщиков: размещение всех сборщиков в одном месте упрощает их поиск для приложения, а также создание аналогичных сборщиков. Наконец, каждая реализация сборщика в `Diamond/src/collectors` находится в своем каталоге (а не в отдельном файле), что позволяет разделить тесты для каждой реализации класса Collector. Также отлично придумано!

Расширяемые пользователем классы (сложное лучше, чем запутанное)

Добавить новую реализацию класса `Collector` нетрудно: нужно создать подкласс абстрактного базового класса `diamond.collector.Collector`[1], реализовать метод `Collector.collect()` и поместить реализацию в отдельный каталог по адресу `venv/src/collectors/`.

Сама по себе реализация сложна, но пользователь этого не знает. В данном разделе рассматриваются простая часть API сборщиков, которая видна пользователю, и сложный код, благодаря которому появился подобный интерфейс.

Сложное против запутанного. Мы можем сравнить работу со сложным кодом со швейцарскими часами — они просто работают, но внутри находится множество

[1] В Python абстрактным считается класс, для которого отдельные методы не определены. Ожидается, что разработчик определит их в подклассе. В абстрактном базовом классе вызов этой функции сгенерирует исключение NotImplementedError. Более современная альтернатива — использовать модуль Python для абстрактных базовых классов (abc) (https://docs.python.org/3/library/abc.html) (впервые реализован в Python 2.6), который сгенерирует ошибку при создании объекта незавершенного класса. Полная спецификация приведена в PEP 3119 (https://www.python.org/dev/peps/pep-3119).

маленьких деталей, взаимодействующих с высокой точностью, чтобы упростить работу с API. Использование запутанного кода похоже на управление самолетом — вы наверняка должны знать, что делать, чтобы не разбиться и не сгореть[1]. Мы не хотим жить в мире без самолетов, но при этом желаем пользоваться часами, не вникая в тонкости их работы. Везде, где это возможно, создавайте менее сложные пользовательские интерфейсы.

Простой пользовательский интерфейс. Для того чтобы создать собственный сборщик данных, пользователь должен создать подкласс абстрактного класса `Collector`, а затем предоставить путь к нему с помощью конфигурационного файла. Рассмотрим пример нового определения класса `Collector` из класса `Diamond/src/collectors/cpu/cpu.py`. Когда Python ищет метод `collect()`, он сначала проверит на наличие `CPUCollector`, а затем, если оно не будет найдено, использует метод `diamond.collector.Collector.collect()`, что сгенерирует исключение `NotImplementedError`.

Код сборщика может выглядеть так:

```
# coding=utf-8
import diamond.collector
import psutil
class CPUCollector(diamond.collector.Collector):
    def collect(self):
        #  В классе Collector содержится лишь инструкция
raise(NotImplementedError)
        metric_name = "cpu.percent"
        metric_value = psutil.cpu_percent()
        self.publish(metric_name, metric_value)
```

Стандартное место для размещения определений сборщиков — каталог `venv/share/diamond/collectors/`, но вы можете хранить их по тому адресу, который укажете в свойстве `collectors_path` конфигурационного файла. Имя класса `CPUCollector` уже указано в примере конфигурационного файла. За исключением добавления спецификаций `hostname` или `hostname_method` в общие стандартные свойства (расположенные под конфигурационным файлом) или в отдельные переопределенные значения для сборщика, как показано в следующем примере, не нужно вносить другие изменения (в документации перечислены дополнительные настройки сборщиков (http://bit.ly/optional-collector-settings)):

```
[[CPUCollector]]
enabled = True
hostname_method = smart
```

Более сложен внутренний код. За кулисами сервер вызовет метод `utils.load_collectors()`, используя путь, указанный в `collectors_path`. Рассмотрим большую часть этой функции (мы сократили ее для удобства).

---

[1] Мы перефразировали отличную статью The Difference Between Complicated and Complex Matters (http://bit.ly/complicated-vs-complex) Ларри Кубана (Larry Cuban), заслуженного профессора Стэнфорда.

```
def load_collectors(paths=None, filter=None):
    """Сканируем путь на предмет сборщиков """
    # Инициализируем возвращаемое значение
    collectors = {}
    log = logging.getLogger('diamond')

    if paths is None:
        return

    if isinstance(paths, basestring):  ❶
        paths = paths.split(',')
        paths = map(str.strip, paths)

    load_include_path(paths)  ❷

    for path in paths:
        ##~~ Пропускаем строки, которые проверяют существование пути
    for f in os.listdir(path):

        # Находимся ли мы в каталоге? Если да, спускаемся вниз по дереву
        fpath = os.path.join(path, f)
        if os.path.isdir(fpath):
            subcollectors = load_collectors([fpath])  ❸
            for key in subcollectors:  ❹
                collectors[key] = subcollectors[key]

            # Игнорируем все, что не является файлом с расширением .py
            elif (os.path.isfile(fpath)
                  ##~~ ... Пропускаем тесты, которые подтверждают,
                  #  что fpath является модулем Python...
                  ):

                ##~~ ... Пропускаем ту часть, которая игнорирует
                #  отфильтрованные пути ...
                modname = f[:-3]

                try:
                    # Импортируем модуль
                    mod = __import__(modname, globals(),
                                     locals(), ['*'])  ❺
                except (KeyboardInterrupt, SystemExit), err:
                    ##~~ ... Записываем исключение в журнал
                    # и завершаем работу ...
                except:
                    ##~~ ... Записываем исключение в журнал
                    # и продолжаем работу...

                # Ищем все классы, определенные в модуле
                for attrname in dir(mod):
                    attr = getattr(mod, attrname)  ❻
```

```
                # Пробуем загружать только те подклассы,
                # которые являются подклассами класса Collector,
                # но не являются базовым классом Collector
                if (inspect.isclass(attr)
                        and issubclass(attr, Collector)
                        and attr != Collector):
                    if attrname.startswith('parent_'):
                        continue
                    # Get class name
                    fqcn = '.'.join([modname, attrname])
                    try:
                        # Загружаем класс Collector
                        cls = load_dynamic_class(fqcn, Collector)
                        # Добавляем класс Collector                    (7)
                        collectors[cls.__name__] = cls
                    except Exception:                    (8)
                        ##~~ Записываем исключение в журнал
                        # и продолжаем работу...

    # Возвращаем классы Collector
    return collectors
```

❶ Разбиваем строку (первый вызов функции); в противном случае пути являются списками строк, содержащих пути, которые указывают места, где реализованы пользовательские подклассы класса `Collector`.

❷ Здесь мы рекурсивно проходим по заданным путям, добавляя каждую папку в `sys.path`, чтобы далее можно было импортировать подклассы класса `Collector`.

❸ Здесь выполняется рекурсия — метод `load_collectors()` вызывает сам себя[1].

❹ После загрузки сборщиков из подкаталогов обновите оригинальный словарь пользовательских сборщиков, добавив туда загруженные сборщики из этих подкаталогов.

❺ С момента введения Python 3.1 модуль `importlib` стандартной библиотеки Python является предпочтительным способом сделать это (с помощью модуля `importlib.import_module`; фрагменты `importlib.import_module` также были портированы в Python 2.7). Это показывает, как можно программно импортировать модуль, используя строку с его именем.

❻ Так можно программно получить доступ к атрибутам модуля, имея лишь строку с именем атрибута.

---

[1] В Python предусмотрен лимит для рекурсии (максимальное количество раз, которое функция может вызвать сама себя) — он по умолчанию значительно ограничивает ваши возможности (существует для того, чтобы предупредить избыточное использование рекурсии). Вы можете узнать значение этого ограничения, введя import sys; sys.getrecursionlimit().

❼ Метод `load_dynamic_class` здесь можно и не использовать. Он повторно импортирует модуль, проверяет, что названный класс является классом на самом деле, проверяет, что он является подклассом класса `Collector`, и, если это верно, — возвращает только что загруженный класс. Избыточность часто встречается в исходном коде, который пишут большие группы людей.

❽ Здесь они получают имя класса для дальнейшего использования при применении настроек из файла конфигурации (имея только строку, содержащую имя класса).

Примеры из стиля Diamond

В Diamond вы можете найти отличный пример использования замыкания, который демонстрирует все, о чем мы говорили в пункте «Замыкания с поздним связыванием» подраздела «Распространенные подводные камни» раздела «Стиль кода» главы 4 по поводу того, что такое поведение зачастую весьма желательно.

Пример использования замыкания (когда подводный камень вовсе не подводный камень). *Замыкание* — это функция, использующая переменные, доступные в локальной области видимости, которые в противном случае будут недоступны при вызове функции. В других языках их, возможно, будет трудно реализовать и понять, но это не относится к Python, поскольку в нем функции обрабатываются так же, как и любые другие объекты[1]. Например, функции могут быть переданы как аргумент, также их можно возвращать из других функций.

Рассмотрим фрагмент кода исполняемого файла `diamond`, который показывает, как реализовать замыкание в Python.

```
##~~ ... Пропускаем операторы импорта ...    ❶

def main():
    try:
        ##~~  ... Пропускаем код, который создает
        ##~~      анализатор командной строки...

        # Анализируем аргументы командной строки
        (options, args) = parser.parse_args()

        ##~~  ... Пропускаем код, который анализирует файл
        ##~~      конфигурации ...
        ##~~  ... Пропускаем код, который настраивает средство
        ##~~      ведения журнала ...
```

[1] Язык программирования имеет функции первого класса, если он рассматривает функции как объекты первого класса. В частности, это означает, что язык поддерживает передачу функций в качестве аргументов другим функциям, возврат их как результат других функций, присваивание их переменным или сохранение в структурах данных.

```
# Передаем сигнал о выходе наверх вместо его обработки
# как обычного исключения
except SystemExit, e:
    raise SystemExit

##~~  ... Пропускаем код, который обрабатывает
##~~      другие исключения, связанные с настройкой ...

try:
    # УПРАВЛЕНИЕ PID ❷
    if not options.skip_pidfile:
        # Инициализируем файл PID
        if not options.pidfile:
            options.pidfile = str(config['server']['pid_file'])

        ##~~ ... Пропускаем код для открытия и чтения
        ##~~     файла PID, если он существует, ...
        ##~~ ... а затем удаления файла, если требуемого PID нет...
        ##~~ ... работа завершается, если процесс уже запущен. ...

    ##~~  ... пропускаем код, который указывает группу
    ##~~      и идентификатор пользователя ...
    ##~~  ... и код, который изменяет разрешения для файла PID. ...

    ##~~  ... пропускаем код, который проверяет, запускать ли демон, ...
    ##~~  ... если демон нужно запускать, процесс открепляется. ...


    # УПРАВЛЕНИЕ PID ❸
    if not options.skip_pidfile:
        # Заканчиваем инициализацию файла PID
        if not options.foreground and not options.collector:
            # Записываем данные в файл PID
            pid = str(os.getpid())
            try:
                pf = file(options.pidfile, 'w+')
            except IOError, e:
                log.error("Failed to write
                          child PID file: %s" % (e))
                sys.exit(1)
            pf.write("%s\n" % pid)
            pf.close()
            # Журналирование
            log.debug("Wrote child PID file:
                      %s" % (options.pidfile))

    # Инициализируем сервер
    server = Server(configfile=options.configfile)

    def sigint_handler(signum, frame): ❹
```

```
            log.info("Signal Received: %d" % (signum))
            # Удаляем файл PID
            if not options.skip_pidfile and
            os.path.exists(options.pidfile):
                os.remove(options.pidfile)
                # Журналирование
                log.debug("Removed PID file: %s" % (options.pidfile)) (5)
            sys.exit(0)

        # Настраиваем обработчики сигналов
        signal.signal(signal.SIGINT, sigint_handler) (6)
        signal.signal(signal.SIGTERM, sigint_handler)

        server.run()

    # Передаем сигнал о выходе наверх вместо его последующей обработки
    # как обычного исключения
    except SystemExit, e:
        raise SystemExit
    ##~~  ... Пропускаем код, который обрабатывает другие исключения ...
    ##~~  ... и остальную часть сценария.
```

(1) Когда мы пропускаем код, отсутствующие части описываем в комментарии, перед которым стоят две тильды (`##~~`).

(2) Мы пользуемся файлом PID[1], чтобы убедиться, что демон уникален (то есть мы не запустили его дважды случайно), а также для быстрого сообщения с другими сценариями при передаче им связанных идентификаторов процессов. Этот файл также нужен для того, чтобы удостовериться, что процесс завершился ненормально (поскольку в этом сценарии файл PID удаляется лишь при нормальном завершении работы).

(3) Весь этот код нужен для того, чтобы предоставить контекст, ведущий к замыканию. К этому моменту либо наш процесс запущен как демон (и теперь имеет другой PID), либо мы пропустим эту часть, поскольку правильный PID уже записан в файл.

(4) `sigint_handler()` и есть замыкание. Оно определяется внутри функции `main()`, а не на высшем уровне, за пределами других функций, поскольку ему нужно знать, искать ли файл PID, и если да — то где.

(5) Позволяет получить информацию из параметров командной строки, которую нельзя получить до вызова функции `main()`. Это означает, что все параметры, связанные с файлом PID, являются локальными переменными пространства имен функции `main`.

(6) Замыкание (функция `sigint_handler()`) отправляется обработчику сигналов; будет использовано для обработки сигналов `SIGINT` и `SIGTERM`.

---

[1] PID расшифровывается как process identifier (идентификатор процесса). Каждый процесс имеет уникальный идентификатор, который доступен в Python благодаря модулю os в стандартной библиотеке: os.getpid().

Tablib

Tablib — это библиотека Python, которая преобразует данные в различные форматы, сохраняет их в объекте класса `Dataset`, а несколько объектов типа `Datasets` — в объекте класса `Databook`. Объекты класса `Dataset` хранятся в форматах JSON, YAML, DBF и CSV (файлы в этих форматах можно импортировать), наборы данных могут быть экспортированы в форматах XLSX, XLS, ODS, JSON, YAML, DBF, CSV, TSV и HTML. Библиотека Tablib выпущена Кеннетом Ритцем (Kenneth Reitz) в 2010 году, имеет интуитивный дизайн API, характерный для всех проектов Ритца.

Читаем небольшую библиотеку

Tablib — это библиотека, а не приложение, поэтому не имеет четко определенной точки входа, как в случае с HowDoI и Diamond.

Загрузите Tablib из GitHub:

```
$ git clone https://github.com/kennethreitz/tablib.git
$ virtualenv -p python3 venv
$ source venv/bin/activate
(venv)$ cd tablib
(venv)$ pip install --editable .
(venv)$ python test_tablib.py  # Run the unit tests.
```

Читаем документацию Tablib

Документация Tablib (http://docs.python-tablib.org/) начинается с упоминания варианта использования, затем в ней более подробно описываются возможности библиотеки: она предоставляет объект типа `Dataset`, который имеет строки, столбцы и заголовки. Вы можете выполнять операции ввода/вывода из разных форматов для объекта типа `Dataset`. В разделе, содержащем более сложные варианты использования, говорится, что вы можете добавлять к строкам теги и создавать унаследованные столбцы, которые являются функциями других столбцов.

Используем Tablib

Tablib — это библиотека, а не исполняемый файл, как в случае с HowDoI или Diamond, поэтому вы можете открыть интерактивную сессию Python и использовать функцию `help()` для исследования API. Рассмотрим пример применения класса `tablib.Dataset`, разных форматов данных и способа работы I/O:

```
>>> import tablib
>>> data = tablib.Dataset()
>>> names = ('Black Knight', 'Killer Rabbit')
>>>
>>> for name in names:
...     fname, lname = name.split()
...     data.append((fname, lname))
...
>>> data.dict
[['Black', 'Knight'], ['Killer', 'Rabbit']]
>>>
>>> print(data.csv)
Black,Knight
Killer,Rabbit
>>> data.headers=('First name', 'Last name')
>>> print(data.yaml)
- {First name: Black, Last name: Knight}
- {First name: Killer, Last name: Rabbit}
>>> with open('tmp.csv', 'w') as outfile:
...     outfile.write(data.csv)
...
64
>>> newdata = tablib.Dataset()
>>> newdata.csv = open('tmp.csv').read()
>>> print(newdata.yaml)
- {First name: Black, Last name: Knight}
- {First name: Killer, Last name: Rabbit}
```

Читаем код библиотеки Tablib

Структура файлов, находящихся в каталоге tablib/, выглядит так:

```
tablib
|--- __init__.py
|--- compat.py
|--- core.py
|--- formats/
|--- packages/
```

Каталоги `tablib/formats/` и `tablib/packages/` рассматриваются в следующих разделах.

Python поддерживает строки документации на уровне модулей, как и строки документации, которые мы уже описали, — строковые литералы, являющиеся первым выражением в функции, классе или методе. На сайте Stack Overflow приведены полезные советы о том, как задокументировать модули (http://stackoverflow.com/a/2557196). Для нас это означает, что существует еще один способ исследовать исходный код — мы можем ввести команду `head *.py` в терминальной оболочке, находясь на верхнем уровне пакета, чтобы показать все строки документации модуля. Вот что мы увидим для библиотеки Tablib:

```
(venv)$ cd tablib
(venv)$ head *.py
==> __init__.py <==    ❶
""" Tablib. """

from tablib.core import (
    Databook, Dataset, detect, import_set, import_book,
    InvalidDatasetType, InvalidDimensions, UnsupportedFormat,
    __version__
)



==> compat.py <==    ❷
# -*- coding: utf-8 -*-

"""
tablib.compat
~~~~~~~~~~~~~

Модуль совместимости Tablib.
"""


==> core.py <==    ❸
# -*- coding: utf-8 -*-
"""
    tablib.core
    ~~~~~~~~~~~

    В этом модуле реализованы основные объекты библиотеки Tablib.

    :copyright: (c) 2014 by Kenneth Reitz.
    :license: MIT, для получения более подробной информации
              смотрите файл LICENSE.
"""
```

Мы узнали следующее.

❶ API высшего уровня (содержимое файла `__init__.py` доступно в каталоге `tablib` после выполнения оператора `import tablib`) имеет всего девять точек входа: классы `Databook` и `Dataset` упоминаются в документации, `detect` может использоваться для определения форматирования, `import_set` и `import_book` должны импортировать данные, а последние три класса — `InvalidDatasetType`, `InvalidDimensions` и `UnsupportedFormat` — выглядят как исключения (если код следует принципам PEP 8, мы можем сказать, какие объекты являются пользовательскими классами на основе их регистра).

❷ `tablib/compat.py` — это модуль совместимости. Беглый взгляд на него покажет, что он обрабатывает проблемы совместимости между Python 2 и Python 3 аналогично

HowDoI, разрешая разные местоположения и имена к одинаковым символам, которые будут использованы в `tablib/core.py`.

❸ В файле `tablib/core.py` в соответствии с его именем реализуются объекты Tablib вроде `Dataset` и `Databook`.

Документация к библиотеке Tablib в формате Sphinx

Документация Tablib (http://docs.python-tablib.org/) содержит хороший пример использования Sphinx (http://www.sphinx-doc.org/en/stable/tutorial.html), поскольку это маленькая библиотека, которая применяет много расширений для Sphinx.

Документация к текущей версии Sphinx находится на странице документации Tablib (http://docs.python-tablib.org/). Если вы хотите построить документацию самостоятельно (пользователям Windows понадобится команда `male` — она старая, но работает как надо), сделайте следующее:

```
(venv)$ pip install sphinx
(venv)$ cd docs
(venv)$ make html
(venv)$ open _build/html/index.html  # Для просмотра результата.
```

Sphinx предоставляет несколько вариантов тем оформления (http://www.sphinx-doc.org/en/stable/theming.html), которые настраиваются с помощью стандартных шаблонов представления кода и стилей CSS. Шаблоны Tablib для левой боковой панели находятся в каталоге `docs/_templates/` в файле `basic/layout.html`. Вы можете найти этот файл в папке стилей Sphinx, введя в командной строке следующую команду:

```
(venv)$ python -c 'import sphinx.themes;print(sphinx.themes.__path__)'
```

Продвинутые пользователи также могут выполнять поиск в `docs/_themes/kr/`, пользовательском стиле, который расширяет базовую структуру. Его можно выбрать, добавив каталог `_themes/` в системный путь, установив необходимые значения свойствам `html_theme_path = ['_themes']` и `html_theme = 'kr'` в `docs/conf.py`.

Для включения в ваш код документации API, которая генерируется автоматически на основе строк документации, используйте `autoclass::`. Вам нужно скопировать форматирование строк документации в Tablib, чтобы это сработало:

```
.. autoclass:: Dataset
   :inherited-members:
```

Для получения этой функциональности следует ответить «да» на вопрос о включении расширения Sphinx autodoc при запуске `sphinx-quickstart`, чтобы создать новый проект Sphinx. Директива `:inherited-members:` также добавит документацию для атрибутов, унаследованных от классов-предков.

Примеры из структуры Tablib

Главная особенность, которую мы хотим подчеркнуть в Tablib, — отсутствие использования классов в модулях в `tablib/formats/` (это идеально иллюстрирует наше утверждение, что не нужно везде применять классы). Далее мы приведем фрагменты кода, демонстрирующие использование синтаксиса декоратора в Tablib, а также задействуем класс `property` (https://docs.python.org/library/functions.html#property) для создания унаследованных атрибутов вроде ширины и высоты набора данных. Помимо этого, покажем, как динамически он регистрирует форматы файлов, чтобы избежать дупликации шаблонного кода для разных типов формата (CSV, YAML и т. д.).

В последних двух подразделах мы рассмотрим, как Tablib использует зависимости, полученные от третьей стороны, а затем обсудим свойство `__slots__` объектов нового класса. Вы можете пропустить эти разделы и при этом продолжать жить счастливой питонской жизнью.

Отсутствие ненужного объектно-ориентированного кода в форматах (использование пространств имен для группирующих функций)

Каталог форматов содержит все определенные для ввода/вывода форматы файлов. Имена модулей `_csv.py`, `_tsv.py`, `_json.py`, `_yaml.py`, `_xls.py`, `_xlsx.py`, `_ods.py` и `_xls.py` начинаются с нижнего подчеркивания — это указывает пользователю библиотеки, что свойства не предназначены для непосредственного использования. Мы можем перейти в каталог `formats` и выполнять поиск классов и функций. Команда `grep ^class formats/*.py` показывает отсутствие определений классов, а команда `grep ^def formats/*.py` — что каждый модуль содержит одну или несколько следующих функций:

- `detect(stream)` определяет формат файла, основываясь на содержимом потока;
- `dset_sheet(dataset, ws)` форматирует клетки для таблиц Excel;
- `export_set(dataset)` экспортирует набор данных в заданный формат, возвращая отформатированную строку в новом формате (для Excel возвращает объект `bytes` или бинарную строку в Python 2);
- `import_set(dset, in_stream, headers=True)` заменяет содержимое набора данных содержимым входного потока;
- `export_book(databook)` экспортирует объекты `Datasheet` в `Databook` в заданном формате, возвращая объект типа `string` или `bytes`;
- `import_book(dbook, in_stream, headers=True)` заменяет содержимое `databook` содержимым входного потока.

Это примеры применения модулей как пространств имен (в конце концов, они же являются отличной штукой) для разделения функций вместо того, чтобы использовать

ненужные классы. Мы узнаем предназначение каждой функции по ее имени, например `formats._csv.import_set()`, `formats._tsv.import_set()` и `formats._json.import_set()` импортируют наборы данных из файлов в формате CSV, TSV и JSON соответственно. Другие функции отвечают за экспорт данных и определение формата файла (где это возможно) для каждого доступного Tablib формата.

Дескрипторы и декораторы свойств (используйте неизменяемость, когда это идет на пользу API)

Tablib — наша первая библиотека, в которой используется синтаксис декораторов Python, описанный в подразделе «Декораторы» раздела «Структурируем проект» главы 4. Синтаксисом предусмотрено указывать символ `@` перед именем функции, вся конструкция размещается над другой функцией. Декоратор изменяет (или декорирует) функцию, которая находится под ним. В следующем фрагменте кода свойство изменяет функции `Dataset.height` и `Dataset.width`, делая их дескрипторами — классами, в которых определен хотя бы один из следующих методов: `__get__()`, `__set__()` или `__delete__()` (геттер, сеттер и метод удаления). Например, поиск атрибута `Dataset.height` приведет к срабатыванию функции-геттера, сеттера или удаления в зависимости от контекста применения атрибута. Такое поведение присуще только новым классам (их мы вскоре обсудим). Для получения более подробной информации о дескрипторах обратитесь к довольно полезному руководству по Python по адресу https://docs.python.org/3/howto/descriptor.html.

```
class Dataset(object):
    #
    # ... опускаем остальную часть определения класса для прозрачности
    #

    @property  (1)
    def height(self):
        """Количество строк в классе :class:`Dataset`.
           Не может быть изменено непосредственно. (2)
        """
        return len(self._data)


    @property
    def width(self):
        """Количество строк в классе :class:`Dataset`.
           Не может быть изменено непосредственно.
        """
        try:
            return len(self._data[0])
        except IndexError:
            try:
                return len(self.headers)
            except TypeError:
                return 0
```

❶ Именно так используется декоратор. В данном случае свойство изменяет `Dataset.height`, чтобы оно вело себя как свойство, а не как связанный метод. Он может работать только с методами классов.

❷ Когда свойство применяется как декоратор, атрибут `height` вернет высоту `Dataset`, но вы не можете задать высоту множества данных, вызвав `Dataset.height`.

Так выглядят атрибуты `height` и `width` при использовании:

```
>>> import tablib
>>> data = tablib.Dataset()
>>> data.header = ("amount", "ingredient")
>>> data.append(("2 cubes", "Arcturan Mega-gin"))
>>> data.width
2
>>> data.height
1
>>>
>>> data.height = 3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: can't set attribute
```

Доступ к `data.height` можно получить так же, как и к любому другому атрибуту, но изменить его значение вы не можете — оно высчитывается на основе данных и всегда актуально. Такой дизайн API весьма эргономичен: конструкцию `data.height` проще ввести на клавиатуре, чем `data.get_height()`; понятно, что означает `data.height`. Поскольку значение этого свойства выводится на основе данных (значение свойства нельзя задать, для него определена только функция-геттер), можно не переживать, что значение свойства рассинхронизируется с реальными данными.

Декоратор свойства можно применить только к атрибутам классов и только к тем классам, которые наследуют от `base object object` (например, `class MyClass(object)`, а не `class MyClass()` — в Python 3 всегда выполняется наследование от объекта).

Один и тот же инструмент используется при создании API для импорта и экспорта данных в Tablib для разных форматов (Tablib не хранит строку для каждого формата). Вместо этого применяются `Dataset`-атрибуты `csv`, `json` и `yaml`, они похожи на свойства `Dataset.height` и `Dataset.width`, показанные в предыдущем примере: вызывают функцию, которая генерирует результат из сохраненных данных или преобразует входной формат и затем заменяет основные данные. Но существует только один набор данных.

Когда свойство `data.csv` находится с левой стороны знака «равно», вызывается функция-сеттер для этого свойства, которая преобразует `dataset` из формата CSV. Когда свойство `data.yaml` находится с правой стороны знака «равно» или стоит отдельно, вызывается функция-геттер для создания строки в заданном формате на основе внутреннего набора данных. Рассмотрим пример.

```
>>> import tablib
>>> data = tablib.Dataset()
>>>
>>> data.csv = "\n".join((    ❶
...     "amount,ingredient",
...     "1 bottle,Ol' Janx Spirit",
...     "1 measure,Santraginus V seawater",
...     "2 cubes,Arcturan Mega-gin",
...     "4 litres,Fallian marsh gas",
...     "1 measure,Qalactin Hypermint extract",
...     "1 tooth,Algolian Suntiger",
...     "a sprinkle,Zamphuor",
...     "1 whole,olive"))
>>>
>>> data[2:4]
[('2 cubes', 'Arcturan Mega-gin'), ('4 litres', 'Fallian marsh gas')]
>>>
>>> print(data.yaml)    ❷
- {amount: 1 bottle, ingredient: Ol Janx Spirit}
- {amount: 1 measure, ingredient: Santraginus V seawater}
- {amount: 2 cubes, ingredient: Arcturan Mega-gin}
- {amount: 4 litres, ingredient: Fallian marsh gas}
- {amount: 1 measure, ingredient: Qalactin Hypermint extract}
- {amount: 1 tooth, ingredient: Algolian Suntiger}
- {amount: a sprinkle, ingredient: Zamphuor}
- {amount: 1 whole, ingredient: olive}
```

❶ Свойство `data.csv`, которое стоит с левой стороны от знака «равно» (оператора присваивания), вызывает функцию `formats.csv.import_set()`, передавая `data` в качестве первого аргумента, и строку, содержащую ингредиенты Пангалактического Грызлодера, в качестве второго аргумента.

❷ Свойство `data.yaml`, стоящее отдельно, вызывает функцию `formats.yaml.export_set()`, передавая `data` в качестве аргумента, выводя строку в формате YAML для функции `print()`.

Функции для получения, установки и удаления данных могут быть привязаны к единому атрибуту с помощью `property`. Его сигнатура выглядит так: `property(fget=None, fset=None, fdel=None, doc=None)`, `fget` определяет функцию-геттер (`formats.csv.import_set()`), `fset` — функцию-сеттер (`formats.csv.export_set()`), а `fdel` — функцию удаления данных (оставлена пустой). Далее мы увидим код, в котором программно устанавливаются свойства форматирования.

Форматы файлов, зарегистрированные программно (не повторяйте дома)

Tablib помещает все подпрограммы для форматирования в подпакет `formats`. Это делает чище основной модуль `core.py` — и целый пакет становится модульным;

добавлять новые форматы файлов будет нетрудно. Несмотря на то что можно копировать фрагменты практически идентичного кода и импортировать поведение при импорте и экспорте для каждого формата отдельно, все форматы *программно* загружаются в свойства, названные в честь каждого формата, в класс `Dataset`.

В следующем примере кода мы выводим все содержимое файла `formats/__init__.py`, поскольку файл не так велик и мы хотим показать, как определяется `formats.available`.

```
# -*- кодировка: utf -8 -*-  ❶

""" Tablib-форматы
"""

from . import _csv as csv
from . import _json as json
from . import _xls as xls
from . import _yaml as yaml
from . import _tsv as tsv
from . import _html as html
from . import _xlsx as xlsx
from . import _ods as ods

available = (json, xls, yaml, csv, tsv, html, xlsx, ods)  ❷
```

❶ В этой строке интерпретатору Python явно указывается, что файл имеет кодировку UTF-8[1].

❷ Определение `formats.available` находится в файле `formats/__init__.py`. Его также можно получить с помощью функции `dir(tablib.formats)`, но приведенный выше список более прост для восприятия.

В файле `core.py` вместо примерно 20 (безобразных и сложных для поддержки) повторяющихся описаний функции для каждого формата код импортирует каждый формат программно, вызывая функцию `self._register_formats()` в конце метода `__init__()` класса `Dataset`. Рассмотрим фрагмент кода, в котором приводится метод `Dataset._register_formats()`.

```
class Dataset(object):
    #
    #  ... опускаем документацию и некоторые определения  ...
    #
```

[1] В Python 2 по умолчанию применяется ASCII, в Python 3 — UTF-8. Существует несколько разрешенных способов взаимодействия между кодировками, все они перечислены в PEP 263 (https://www.python.org/dev/peps/pep-0263/). Вы можете использовать тот, который лучше работает в вашем любимом текстовом редакторе.

```
    @classmethod  ❶
    def _register_formats(cls):
        """Добавляем свойства форматов."""
        for fmt in formats.available:  ❷
            try:
                try:
                    setattr(cls, fmt.title,
                    property(fmt.export_set, fmt.import_set))  ❸
                except AttributeError:  ❹
                    setattr(cls, fmt.title, property(fmt.export_set)) ❺
                except AttributeError:
                    pass  ❻

    #
    # ... пропускаем другие определения ...
    #

    @property  ❼
    def tsv():
        """Представление TSV объекта :class:`Dataset`. Верхний ряд
        будет содержать заголовки, если они были установлены.
        В противном случае верхний ряд будет содержать
        первый ряд набора данных.

        Объект dataset также может быть импортирован путем установки
        атрибута :class:`Dataset.tsv`. ::

            data = tablib.Dataset()
            data.tsv = 'age\tfirst_name\tlast_name\n90\tJohn\tAdams'  ❽

        Import assumes (for now) that headers exist.
        """

        pass
```

❶ Символ `@classmethod` является декоратором (они подробно описаны в подразделе «Декораторы» подраздела «Структурируем проект» главы 4). Декоратор модифицирует метод `_register_formats()` таким образом, что он начинает передавать в качестве первого аргумента класс объекта (`Dataset`), а не его экземпляр (`self`).

❷ Параметр `formats.available` определен в файле `formats/__init__.py` и содержит все доступные форматы.

❸ В этой строке `setattr` присваивает значение атрибуту с именем `fmt.title` (то есть `Dataset.csv` или `Dataset.xls`). Это значение особенное: функция `property(fmt.export_set, fmt.import_set)` превращает `Dataset.csv` в *свойство*.

❹ Если свойство `fmt.import_set` не будет определено, возникнет исключение `AttributeError`.

❺ Если функции импорта нет, попробуйте присвоить лишь поведение экспорта.

❻ Если нет ни функции импорта, ни функции экспорта, не присваивайте ничего.

❼ Каждый из форматов файлов определен здесь как свойство, имеет описательную строку документации. Строка документации будет сохранена, когда функция `property()` будет вызвана в точке ❸ или ❺ для присвоения дополнительного поведения.

❽ `\t` и `\n` — управляющие последовательности, которые представляют собой, соответственно, символ табуляции и новую строку. Все они перечислены в документации к строковым литералам Python (https://docs.python.org/3/reference/lexical_analysis.html#index-18).

Но мы все — ответственные пользователи

Эти способы использования декоратора `@property` *не* похожи на способы применения аналогичных инструментов в Java, цель которых состоит в том, чтобы управлять доступом пользователей к данным. Это идет вразрез с философией Python, которая гласит, что *мы все — ответственные пользователи*. Цель применения декоратора `@property` — отделение данных от функций просмотра, связанных с данными (в этом случае с высотой, шириной и разными форматами хранения). В ситуации, когда геттеры и сеттеры не нужны для предобработки или постобработки, более питонским вариантом поведения будет присвоение данных обычному атрибуту и разрешение пользователю взаимодействовать с ними.

Зависимости, полученные от третьей стороны, в пакетах (пример их использования)

Зависимости Tablib в данный момент поставляются с кодом — в каталоге `packages`, но могут в будущем быть перемещены в систему надстроек. Каталог `packages` содержит сторонние пакеты, используемые внутри Tablib, чтобы гарантировать совместимость; другой вариант — указание версий в файле `setup.py`, который будет загружен и установлен в момент установки Tablib. Этот прием рассматривается в разделе «Зависимости, получаемые от третьей стороны» раздела «Структурируем проект» главы 4. Для Tablib был выбран вариант поведения, позволяющий снизить количество зависимостей, который нужно загружать пользователям, и поскольку иногда для Python 2 и Python 3 требуются разные пакеты, в этом случае включаются оба пакета. (Соответствующий пакет импортируется, функции вызываются с помощью их обычного имени в файле `tablib/compat.py`.) Таким образом, Tablib может иметь одну базу кода вместо двух — по одной для каждой версии Python. Раз каждая из зависимостей имеет собственную лицензию, на верхний уровень каталога проекта был добавлен документ NOTICE, в котором перечисляются лицензии каждой зависимости.

Экономим память с помощью свойства __slots__ (оптимизируйте с осторожностью)

Скорости Python предпочитает читаемость. Его дизайн, афоризмы, из которых состоит его дзен, и раннее влияние, которое на него оказали языки вроде ABC (http://bit.ly/abc-to-python), — все это заставляет ставить дружелюбие к пользователю над производительностью (более подробно об оптимизации мы поговорим в разделе «Скорость» главы 8).

Использование свойства `__slots__` в Tablib — этот тот случай, когда оптимизация имеет значение. Данная ссылка выглядит несколько странно, она доступна только для новых классов (они описаны через несколько страниц), но мы хотим показать, что при необходимости вы можете оптимизировать Python.

Подобная оптимизация полезна только в том случае, если у вас имеется много небольших объектов, поскольку она сократит отпечаток каждого объекта класса на размер одного словаря (крупные объекты сделают такую небольшую экономию нерелевантной, а для малого количества объектов такая экономия не стоит затраченных усилий).

Рассмотрим фрагмент из документации `__slots__` (http://bit.ly/__slots__-doc).

> По умолчанию объекты классов имеют словарь для хранения атрибутов. Он занимает слишком много места, если объекты имеют малое количество переменных объекта. Использование места может стать особенно заметным при создании большого количества объектов.
>
> Ситуацию можно изменить путем объявления __slots__ в описании класса. Объявление __slots__ принимает последовательность переменных объекта и резервирует достаточный объем памяти для каждой переменной. Место экономится, поскольку для каждой переменной теперь не создается __dict__.

Обычно вам не следует об этом беспокоиться: обратите внимание, что свойство `__slots__` не появляется в классах `Dataset` или `Databook` — только в классе `Row`, но поскольку рядов данных может быть очень много, использование `__slots__` выглядит хорошим решением. Класс `Row` не показан в `tablib/__init__.py`, поскольку является вспомогательным классом для класса `Dataset`, для каждой строки создается один объект такого класса.

Рассмотрим, как выглядит определение `__slots__` в самом начале определения класса `Row`:

```
class Row(object):
    """Внутренний объект Row. Используется в основном для фильтрации."""
    __slots__ = ['_row', 'tags']
    def __init__(self, row=list(), tags=list()):
```

```
        self._row = list(row)
        self.tags = list(tags)
    #
    #  ... и т.д. ...
    #
```

Проблема в том, что больше не существует атрибута `__dict__`, в котором хранятся объекты класса `Row`, но функция `pickle.dump()` (вызывается для сериализации объектов) по умолчанию использует `__dict__` для сериализации объектов, если только не определен метод `__getstate__()`. Аналогично во время десериализации (процесса, который читает сериализованные байты и восстанавливает объект в памяти), если метод `__setstate__()` не определен, метод `pickle.load()` загружает данные в атрибут объекта `__dict__`. Рассмотрим, как это обойти.

```
class Row(object):
    #
    #  ... пропускаем другие определения ...
    #
    def __getstate__(self):
        slots = dict()
        for slot in self.__slots__:
            attribute = getattr(self, slot)
            slots[slot] = attribute
        return slots
    def __setstate__(self, state):
        for (k, v) in list(state.items()):
            setattr(self, k, v)
```

Для получения более подробной информации о методах `__getstate__()` и `__setstate__()` и сериализации обратитесь к документации `__getstate__` (http://bit.ly/__getstate__-doc).

Примеры из стиля Tablib

Мы подготовили лишь один пример использования стиля в Tablib — перегрузка операторов (это позволяет рассмотреть детали модели данных Python). Настройка поведения ваших классов позволит разработчикам, использующим ваш API, писать хороший код.

Перегрузка операторов (красивое лучше, чем уродливое). В этом фрагменте кода приводится перегрузка операторов Python, чтобы выполнять операции для строк или столбцов набора данных. В первом фрагменте кода показывается интерактивное применение квадратных скобок [ ] как для численных индексов, так и для имен столбцов, а во втором — код, который использует это поведение.

```
>>> data[-1]  ❶
('1 whole', 'olive')
>>>
>>> data[-1] = ['2 whole', 'olives']  ❷
```

```
>>>
>>> data[-1]
('2 whole', 'olives')  ❸
>>>
>>> del data[2:7]  ❹
>>>
>>> print(data.csv)
amount,ingredient  ❺
1 bottle,Ol' Janx Spirit
1 measure,Santraginus V seawater
2 whole,olives

>>> data['ingredient']  ❻
["Ol' Janx Spirit", 'Santraginus V seawater', 'olives']
```

❶ Если вы указываете в квадратных скобках числа, этот оператор вернет строку, которая находится в заданной позиции.

❷ Благодаря этому оператору присваивания с квадратными скобками...

❸ ... вместо исходной одной оливки у вас становится две.

❹ Здесь выполняется удаление с помощью *вырезки* — 2:7 указывает на числа 2, 3, 4, 5, 6, но не 7.

❺ Взгляните, насколько сокращается рецепт после выполнения.

❻ Вы также можете получить доступ к столбцам с помощью имени.

В части кода класса `Dataset`, которая определяет поведение оператора «квадратные скобки», показывается, как обрабатывать доступ по имени столбца и номеру строки.

```
class Dataset(object):
    #
    #  ... пропускаем часть определений для краткости ...
    #
    def __getitem__(self, key):
        if isinstance(key, str) or isinstance(key, unicode):  ❶
             if key in self.headers:
                  pos = self.headers.index(key)  ❷
                  # получаем индекс 'key' из каждого объекта данных
                  return [row[pos] for row in self._data]
             else:  ❸
                raise KeyError
        else:
            _results = self._data[key]
            if isinstance(_results, Row):  ❹
                return _results.tuple
            else:
                return [result.tuple for result in _results]  ❺
```

```
def __setitem__(self, key, value):  ❻
    self._validate(value)
    self._data[key] = Row(value)


def __delitem__(self, key):
    if isinstance(key, str) or isinstance(key, unicode):  ❼
        if key in self.headers:
            pos = self.headers.index(key)
            del self.headers[pos]

            for row in self._data:
                del row[pos]
        else:
            raise KeyError
    else:
        del self._data[key]
```

❶ Во-первых, проверим, что именно мы ищем — столбец (`True`, если `key` является строкой) или строку (`True`, если `key` является числом или вырезкой).

❷ Этот код проверяет наличие ключа в `self.headers` и затем...

❸ ...явно вызывает исключение `KeyError`, поэтому, если вы получаете доступ по имени столбца, поведение будет таким же, как и у словаря. Весь блок `if/else` необязателен для работы функции — если его опустить, исключение `ValueError` будет сгенерировано функцией `self.headers.index(key)` в том случае, если ключа нет в `self.headers`. Единственным его предназначением является предоставление пользователю библиотеки более информативной ошибки.

❹ Здесь определяется, чем является ключ — числом или вырезкой (вроде 2:7). Если вырезкой, `_results` будет списком, а не объектом класса `Row`.

❺ Здесь обрабатывается вырезка. Поскольку строки возвращаются как кортежи, значения представляют собой неизменяемые копии реальных данных — и данные из набора (которые хранятся в списках) не будут случайно повреждены в результате присваивания.

❻ Метод `__setitem__()` может изменить одну строку, но не столбец. Это сделано намеренно, так как не существует способа изменить содержимое всего столбца; с точки зрения целостности данных такое решение не самый плохой выбор. Пользователь всегда может преобразовать столбец и внедрить его в любую позицию с помощью одного из методов `insert_col()`, `lpush_col()` или `rpush_col()`.

❼ Метод `__delitem__()` удалит столбец или строку, используя ту же логику, что и метод `__getitem__()`.

Для получения более подробной информации о перегрузке операторов и других особых методах смотрите документацию Python о специальных именах методов (http://bit.ly/special-method-names).

Requests

В День святого Валентина в 2011 году Кеннет Ритц (Kenneth Reitz) опубликовал «любовное письмо», адресованное сообществу Python, — библиотеку Requests. Она была воспринята с большим энтузиазмом благодаря интуитивно понятному дизайну API (это значит, что API настолько прост и понятен, что вам почти не нужна документация для того, чтобы им пользоваться).

Читаем более крупную библиотеку

Библиотека Requests намного крупнее библиотеки Tablib и имеет множество модулей. Однако мы подходим к вопросу ее прочтения точно так же — просмотрим документацию и будем следовать API в коде.

Загрузите Requests из GitHub:

```
$ git clone https://github.com/kennethreitz/requests.git
$ virtualenv -p python3 venv
$ source venv/bin/activate
(venv)$ cd requests
(venv)$ pip install --editable .
 (venv)$ pip install -r requirements.txt  # Required for unit tests
(venv)$ py.test tests  # Run the unit tests.
```

Некоторые тесты могут дать сбой. Например, если ваш интернет-провайдер перехватывает ошибку 404 для того, чтобы показать вам какую-нибудь рекламу, вы не сможете сгенерировать исключение `ConnectionError`.

Читаем документацию библиотеки Requests

Библиотека Requests находится в гораздо более крупном пакете, поэтому сначала просмотрите лишь заголовки разделов в документации к Requests (http://docs.python-requests.org/). Requests расширяет библиотеки urrlib и httplib, которые вы можете найти в стандартной библиотеке Python, предоставляя методы, выполняющие запросы HTTP. Библиотека предусматривает поддержку международных доменов и URL, автоматическую декомпрессию, автоматическое декодирование содержимого, проверку сертификатов SSL, поддержку прокси для HTTP(S) и другую функциональность, определенную стандартами Internet Engineering Task Force (IETF) для HTTP с помощью запросов комментария (requests for comment, RFC) 7230-7235[1].

[1] Если вам нужно обновить свои знания о словаре, воспользуйтесь RFC 7231 — документом, содержащим семантику HTTP (http://bit.ly/http-semantics). Если вы просмотрите его содержимое и прочтете введение, то сможете понять, упоминается ли нужное вам определение, и найти его.

Библиотека Requests стремится покрыть все спецификации HTTP от IETF, задействуя всего несколько функций, набор аргументов с ключевым словом и несколько классов.

Используем Requests

Как и в случае Tablib, в строках документации содержится достаточно информации для того, чтобы использовать Requests, не обращаясь к документации, размещенной онлайн. Рассмотрим следующий пример:

```
>>> import requests
>>> help(requests)  # Показывает руководство по использованию
                    # и указывает искать 'requests.api'
>>> help(requests.api)  # Показывает подробное описание API
>>>
>>> result = requests.get('https://pypi.python.org/pypi/requests/json')
>>> result.status_code
200
>>> result.ok
True
>>> result.text[:42]
'{\n    "info": {\n        "maintainer": null'
>>>
>>> result.json().keys()
dict_keys(['info', 'releases', 'urls'])
>>>
>>> result.json()['info']['summary']
'Python HTTP for Humans.'
```

Читаем код Requests

Рассмотрим содержимое пакета Requests.

```
$ ls
__init__.py      cacert.pem ❶    exceptions.py    sessions.py
adapters.py      certs.py        hooks.py         status_codes.py
api.py           compat.py       models.py        structures.py
auth.py          cookies.py      packages/ ❷      utils.py
```

❶ `cacert.pem` — стандартный набор сертификатов, который используется при проверке сертификатов SSL.

❷ Requests имеет простую структуру, за исключением каталога `packages`, хранящего сторонние зависимости `chardet` и `urllib3`. Они импортируются как `requests.packages.chardet` и `requests.packages.urllib3`, поэтому программисты все еще могут получить доступ к `chardet` и `urllib3` из стандартной библиотеки.

Мы можем разобраться в происходящем благодаря удачно выбранным именам модулей, но если нужно больше информации, просмотрите строки документации

модуля, введя `head *.py` в каталоге верхнего уровня. В следующем списке эти строки документации приводятся в сокращенном виде (не показывается `compat.py`). Исходя из его имени (он назван так же, как и аналогичный файл библиотеки Reitz's Tablib, мы можем сделать вывод, что он отвечает за совместимость между Python 2 и Python 3).

- `api.py` — реализует Requests API.
- `hooks.py` — предоставляет возможность использовать систему функций перехвата Requests.
- `models.py` — содержит основные объекты, которыми пользуется Requests.
- `sessions.py` — предоставляет объект `Session` для управления настройками и их сохранения между запросами (cookies, авторизация, прокси).
- `auth.py` — содержит дескрипторы для аутентификации в Requests.
- `status_codes.py` — таблица, в которой соотносятся заголовки состояний и их коды.
- `cookies.py` — код для совместимости, который позволяет использовать `cookielib.CookieJar` с запросами.
- `adapters.py` — содержит транспортные адаптеры, которые Requests применяет для определения и поддержания соединений.
- `exceptions.py` — все исключения Requests.
- `structures.py` — структуры данных, которыми пользуется Requests.
- `certs.py` — возвращает предпочтительный набор сертификатов CA по умолчанию, в котором перечислены доверенные сертификаты SSL.
- `utils.py` — предоставляет вспомогательные функции, которые используются внутри Requests и могут применяться внешними пользователями.

Что мы узнали из заголовков:

- существует система функций перехвата (hooks.py) — это подразумевает, что пользователь может модифицировать способ работы библиотеки Requests. Мы не будем обсуждать этот вопрос подробно, чтобы не отвлекаться от темы;
- основным модулем является `models.py`, поскольку в нем содержатся «основные объекты, которыми пользуется Requests»;
- основная причина существования `sessions.Session` — сохранение cookies между несколькими запросами (например, это может понадобиться во время аутентификации);
- соединение HTTP создается с помощью объектов из модуля adapters.py;
- остальная часть проекта довольно очевидна: `auth.py` нужен для аутентификации, `status_codes.py` содержит коды состояний, `cookies.py` нужен для добавления и удаления cookies, `exceptions.py` — для исключений, `structures.py` содержит структуры данных (например, не зависящий от регистра словарь), `utils.py` — вспомогательные функции.

Идея поместить модуль коммуникации в отдельный файл `adapters.py` инновационна (во всяком случае для этого разработчика). Это означает, что `models.Request`, `models.PreparedRequest` и `models.Response` на самом деле ничего не делают — просто сохраняют данные, возможно несколько изменяя их в угоду представлению, сериализации или кодировке. Действия обрабатываются отдельными классами, которые существуют только для того, чтобы выполнить, например, аутентификацию или коммуникацию. Каждый класс делает что-то одно, и каждый модуль содержит классы, выполняющие похожие задачи, — в этом и проявляется питонский подход, который многие из нас используют для определений функций.

Строки документации Requests, совместимые со Sphinx

Если вы начинаете новый проект и используете Sphinx и его расширения autodoc, вам нужно отформатировать строки документации таким образом, чтобы Sphinx смог проанализировать их.

В документации Sphinx не всегда получится легко найти нужные ключевые слова. Многие рекомендуют копировать строки документации в Requests, если вы хотите, чтобы формат был правильным, а не искать инструкции в документации Sphinx. Например, рассмотрим определение функции `delete()` в файле `requests/api.py`:

```
def delete(url, **kwargs):
    """Отправляет запрос DELETE.
    :param url: URL для нового объекта :class:'Request'.
    :param \*\*kwargs: Необязательные аргументы, которые принимает
                       ''request''.
    :return: объект класса :class:'Response <Response>'
    :rtype: requests.Response
    """
    return request('delete', url, **kwargs)
```

Представление этого определения в Sphinx autodoc находится в онлайн-документации к API (http://docs.python-requests.org/en/master/api/#requests.delete).

Примеры из структуры Requests

Все любят API Requests — его просто запомнить, он помогает пользователям писать простой и красивый код. В этом разделе сначала рассматривается предпочтительный дизайн для более полных сообщений об ошибках и запоминающийся API, который, как мы думаем, привел к созданию модуля `requests.api`. Затем мы рассмотрим разницу между объектами `requests.Request` и `urllib.request.Request` и расскажем, для чего существует объект `requests.Request`.

Высокоуровневый API (очевидный способ решить задачу, желательно единственный)

Функции, определенные в файле `api.py` (за исключением `request()`), названы в честь методов запросов HTTP[1]. Все методы запроса практически одинаковы, за исключением имени метода и возможных параметров с ключевыми словами, поэтому мы удалим из этого фрагмента весь код, расположенный в файле `requests/api.py` после функции `get()`.

```
# -*- coding: utf-8 -*-
"""

requests.api
~~~~~~~~~~~~

Этот модуль реализует Requests API.

:copyright: (c) 2012 by Kenneth Reitz.
:license: Apache2, see LICENSE for more details.
"""

from . import sessions

def request(method, url, **kwargs):  (1)
    """Создает и отправляет :class:`Request <Request>`.

    :param method: метод для нового объекта класса :class:`Request`.
    :param url: URL для нового объекта :class:`Request`.
    :param params: (необязательный) словарь или объект
     типа bytes, который будет отправлен в строке запроса
     для класса :class:`Request`.

    ... пропускаем документацию для оставшихся аргументов  (2)
    с ключевым словом ...

    :return: :class:`Response <Response>` object
    :rtype: requests.Response

    Usage::

        >>> import requests
        >>> req = requests.request('GET', 'http://httpbin.org/get')
        <Response [200]>
    """
```

[1] Они определены в разделе 4.3 текущего рабочего предложения для HTTP (http://bit.ly/http-method-def).

```
    # Используя оператор 'with', мы гарантируем, что сессия закрыта,
    # тем самым мы избегаем забытых открытых сокетов,
    # которые могут вызвать ResourceWarning
    # в одних случаях и выглядеть как утечка памяти в других.
    with sessions.Session() as session:  ❸
        return session.request(method=method, url=url, **kwargs)

def get(url, params=None, **kwargs):  ❹
    """Отправляет запрос GET.

    :param url: URL для нового объекта new :class:`Request`.
    :param params: (необязательный) объект типа Dictionary или bytes,
                   который должен быть отправлен
                   в строке запроса для :class:`Request`.
    :param \*\*kwargs: необязательные аргументы,
                       которые принимает ``request``.
    :return: объект класса :class:`Response <Response>`
    :rtype: requests.Response
    """

    kwargs.setdefault('allow_redirects', True)  ❺
    return request('get', url, params=params, **kwargs)  ❻
```

❶ Функция `request()` содержит в своей сигнатуре `**kwargs`. Это означает, что дополнительные аргументы с ключевым словом не сгенерируют исключение, также это скрывает параметры от пользователя.

❷ В документации, опущенной в этом фрагменте для краткости, описывается каждый аргумент с ключевым словом, с которым связано действие. Использование `**kwargs` из сигнатуры вашей функции — единственный способ для пользователя сказать, каким должно быть содержимое `**kwargs`, не заглядывая в код.

❸ С помощью оператора `with` Python поддерживает контекст времени выполнения. Оно может быть использовано для любого объекта, для которого определены методы `__enter__()` и `__exit__()`. Метод `__enter()__` будет вызван при входе в оператор `with`, а `__exit__()` — при выходе, независимо от того, завершился оператор нормально или сгенерировал исключение.

❹ Функция `get()` получает ключевое слово `params=None`, применяя значение по умолчанию `None`. Аргумент с ключевым словом `params` важен для `get`, поскольку будет использоваться в строке запроса HTTP. Предоставление отдельных аргументов с ключевым словом дает гибкость действий опытным пользователям (благодаря оставшимся `**kwargs`), упрощая работу для 99 % людей, которым это не нужно.

❺ По умолчанию функция `request()` не разрешает перенаправление, поэтому на этом шаге устанавливается значение `True`, если пользователь не сделал этого заранее.

❻ Функция `get()` вызывает функцию `request()`, передавая в качестве первого параметра `get`. Создание функции `get` имеет два преимущества перед использованием строкового аргумента вроде `request("get", ...)`. Во-первых, даже без документации становится очевидно, какие методы HTTP доступны в этом API. Во-вторых, если пользователь сделает опечатку в имени метода, исключение `NameError` будет сгенерировано быстрее, и, возможно, его будет проще отследить, чем если бы оно было сгенерировано более глубоко в коде.

В файле `requests/api.py` нет новой функциональности; она существует для того, чтобы предоставить пользователю простой API. Плюс размещение строковых методов HTTP непосредственно в API в качестве имен функций означает, что любая опечатка в имени метода будет найдена на ранних этапах, например:

```
>>> requests.foo('http://www.python.org')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'module' object has no attribute 'foo'
>>>
>>> requests.request('foo', 'http://www.python.org')
<Response [403]>
```

Объекты класса Request и PreparedRequest (мы все — ответственные пользователи)

Файл `__init__.py` предоставляет классы `Request`, `PreparedRequest` и `Response` из файла `models.py` как часть основного API. Зачем вообще нужен файл `models.Request`? В стандартной библиотеке уже существует `urllib.requests.Request`, и в файле `cookies.py` находится объект `MockRequest`, который оборачивает `models.Request`, чтобы он работал как `urllib.requests.Request` для `http.cookiejar`[1]. Это означает, что любые методы, необходимые для взаимодействия объекта типа `Request` с библиотекой cookies, намеренно исключены из `requests.Request`. Для чего эти лишние усилия?

Дополнительные методы в `MockRequest` (нужен для эмуляции `urllib.request.Request` для библиотеки cookies) используются библиотекой cookies для управления cookies. За исключением функции `get_type()` (которая обычно возвращает `http` или `https` при использовании) и непроверяемого свойства (в нашем случае `True`), они связаны с URL или заголовками запросов.

Связанные с заголовками:

- `add_unredirected_header()` — добавить в заголовок новую пару ключ-значение;
- `get_header()` — получить определенное имя из словаря заголовков;

[1] Модуль http.cookiejar ранее назывался cookielib в Python 2, urllib.requests.Request ранее назывался urllib2.Request в Python 2.

- `get_new_headers()` — получить словарь, содержащий новые заголовки (которые добавлены с помощью `cookielib`);
- `has_header()` — проверяем, существует ли имя в словаре заголовков.

Связанные с URL:

- `get_full_url()` — соответствует своему имени;
- `host и origin_req_host` — свойства, чьи значения устанавливаются путем вызова методов `get_host()` и `get_origin_req_host()` соответственно;
- `get_host()` — извлекает хост из URL (например, www.python.org из https://www.python.org/dev/peps/pep-0008/);
- `get_origin_req_host()` — вызывает `get_host()`[1].

Все они являются функциями доступа, за исключением `MockRequest.add_unredirected_header()`.

В строке документации к объекту `MockRequest` указывается, что «оригинальный объект запроса доступен только для чтения».

В `requests.Request` вместо этого непосредственно доступны атрибуты данных. Это делает все функции доступа ненужными: для получения или установки заголовков требуется лишь получить доступ к словарю `request-instance.headers`. Аналогично пользователь может получить или изменить строку URL: `request-instance.url`.

Объект `PreparedRequest` инициализируется пустым и заполняется при вызове метода `prepared-request-instance.prepare()`, что наполняет его релевантными данными (обычно получаемыми путем вызова объекта `Request`). В этот момент применяются исправления регистра символов, кодировки и пр. Содержимое объекта после подготовки можно будет отправить на сервер, но к каждому атрибуту все еще можно получить доступ непосредственно. Доступен даже `PreparedRequest._cookies`, однако нижнее подчеркивание, с которого начинается это имя, намекает на то, что атрибут не предназначен для использования за пределами класса, не запрещая при этом доступ (мы все — ответственные пользователи).

Этот выбор позволяет пользователю изменять объекты, но они становятся гораздо читабельнее, а дополнительная работа, выполняемая внутри `PreparedRequest`, позволяет исправить проблемы с регистром и использовать словарь вместо `CookieJar` (ищите оператор `if isinstance()/else`):

```
#
#  ... из файла models.py ...
#
class PreparedRequest():
```

[1] Этот метод позволяет обрабатывать запросы из разных источников (вроде получения библиотеки JavaScript, размещенной на стороннем сайте). Он должен возвращать изначальный хост запроса, определенный в IETF RFC 2965.

```
    #
    #  ...пропускаем все остальное...
    #
    def prepare_cookies(self, cookies):
        """Подготавливает данные заданного HTTP cookie.
        Эта функция в итоге генерирует заголовок ''Cookie'' на основе
        предоставленных cookies с использованием cookielib. Из-за особенностей
        дизайна cookielib заголовок не будет сгенерирован повторно,
        если он уже существует. Это значит, что эта функция может быть
        вызвана всего один раз во время жизни объекта
        :class:'PreparedRequest <PreparedRequest>'. Любые последующие вызовы
        ''prepare_cookies'' не возымеют эффекта, если только заголовок "Cookie"
        не будет удален заранее."""
        if isinstance(cookies, cookielib.CookieJar):
            self._cookies = cookies
        else:
            self._cookies = cookiejar_from_dict(cookies)
        cookie_header = get_cookie_header(self._cookies, self)
        if cookie_header is not None:
            self.headers['Cookie'] = cookie_header
```

Эти детали могут показаться незначительными, но они позволяют создать интуитивно понятный API.

Примеры из стиля Requests

Примеры стиля из Requests демонстрируют использование множеств (по нашему мнению, о них незаслуженно забывают!), мы также взглянем на модуль `requests.status_codes module` — он задействуется для упрощения стиля остального кода и позволяет избежать применения жестко закодированных кодов состояния HTTP в остальных местах.

Множества и их арифметика (отличная питонская идиома)

Мы еще не приводили пример использования множеств в Python. Множества в Python ведут себя так же, как и множества в математике: вы можете выполнить операции вычитания, объединения (с помощью оператора ИЛИ) и пересечения (с помощью оператора И):

```
>>> s1 = set((7,6))
>>> s2 = set((8,7))
>>> s1
{6, 7}
>>> s2
{8, 7}
>>> s1 - s2  # разность множеств
{6}
```

```
>>> s1 | s2  # объединение множеств
{8, 6, 7}
>>> s1 & s2  # пересечение множеств
{7}
```

Рассмотрим пример работы с множествами, вы можете найти его в конце этой функции из файла `cookies.py` (рядом с пометкой ❷):

```
#
# ... из файла cookies.py ...
#

def create_cookie(name, value, **kwargs):  ❶
    """Создаем cookie на основе указанных параметров.
    По умолчанию пара name — value будет установлена
    для домена '' и отправлена при каждом запросе
    (это иногда называется "supercookie").
    """

    result = dict(
        version=0,
        name=name,
        value=value,
        port=None,
        domain='',
        path='/',
        secure=False,
        expires=None,
        discard=True,
        comment=None,
        comment_url=None,
        rest={'HttpOnly': None},
        rfc2109=False,)

    badargs = set(kwargs) - set(result)  ❷
    if badargs:
        err = 'create_cookie() got unexpected  ❸
              keyword arguments: %s'
        raise TypeError(err % list(badargs))

    result.update(kwargs)  ❹
    result['port_specified'] = bool(result['port'])  ❺
    result['domain_specified'] = bool(result['domain'])
    result['domain_initial_dot'] = result['domain'].startswith('.')
    result['path_specified'] = bool(result['path'])

    return cookielib.Cookie(**result)  ❻
```

❶ Спецификация `**kwargs` позволяет пользователю предоставить любой параметр с ключевым словом для cookie (или не предоставлять их вовсе).

❷ Арифметика множеств! Питонская. Простая. Доступная в стандартной библиотеке. Для словаря функция `set()` формирует множество ключей.

❸ Это отличный пример того, что разбиение длинной строки на две короткие более разумно. Дополнительная переменная err не нанесла никакого вреда.

❹ Вызов `result.update(kwargs)` обновляет словарь `result` парами ключ/значение, содержащимися в словаре `kwargs`, заменяя существующие пары или создавая те пары, которых не было.

❺ Здесь вызов метода `bool()` возвращает значение `True`, если объект верен (это значит, что его значение оценивается как `True` — в данном случае вызов `bool(result['port'])` оценивается как `True`, если значение не равно `None` и не является пустым контейнером).

❻ Сигнатура для инициализации `cookielib.Cookie` представляет собой набор из 18 позиционных аргументов и одного аргумента с ключевым словом (`rfc2109` по умолчанию считается равным `False`). Нам, как среднестатистическим пользователям, невозможно запомнить все значения и их позиции, поэтому Requests позволяет присваивать значения позиционным аргументам, основываясь на их имени, как в случае аргументов с ключевым словом, отправляя целый словарь.

Коды состояний (читаемость имеет значение)

Файл `status_codes.py` нужен только для того, чтобы создать объект, который может искать коды состояний по атрибуту. Мы сначала покажем определение словаря в файле `status_codes.py`, а затем — фрагмент кода файла `sessions.py`, в котором словарь используется.

```
#
#  ... фрагмент файла requests/status_codes.py ...
#

_codes = {

    # Справочная информация.
    100: ('continue',),
    101: ('switching_protocols',),
    102: ('processing',),
    103: ('checkpoint',),
    122: ('uri_too_long', 'request_uri_too_long'),
    200: ('ok', 'okay', 'all_ok', 'all_okay',
          'all_good', '\\o/', '✔'),   ❶
    201: ('created',),
    202: ('accepted',),
    #
    #  ... пропускаем ...
    #
```

```
    # Перенаправление.
    300: ('multiple_choices',),
    301: ('moved_permanently', 'moved', '\\o-'),
    302: ('found',),
    303: ('see_other', 'other'),
    304: ('not_modified',),
    305: ('use_proxy',),
    306: ('switch_proxy',),
    307: ('temporary_redirect', 'temporary_moved', 'temporary'),
    308: ('permanent_redirect',
    'resume_incomplete', 'resume',),
    # Эти два будут удалены в версии 3.0   ❷

    #
    #  ... пропускаем остальные ...
    #
}

codes = LookupDict(name='status_codes')   ❸

for code, titles in _codes.items():
    for title in titles:
        setattr(codes, title, code)   ❹
        if not title.startswith('\\'):
            setattr(codes, title.upper(), code)   ❺
```

❶ Все эти варианты состояния `OK` станут ключами словаря. За исключением счастливого человека (\\o/) и флажка (✓).

❷ Устаревшие значения расположены на отдельной строке, поэтому, когда их в будущем удалят, в системе контроля версий это будет четко определено.

❸ `LookupDict` позволяет получать доступ к своим элементам через точку, как это показано в следующей строке.

❹ `codes.ok == 200` и `codes.okay == 200`.

❺ А также `codes.OK == 200` и `codes.OKAY == 200`.

Вся эта работа нужна для того, чтобы создать словарь с кодами состояний. Для чего? Использование словаря вместо цифр на протяжении всего кода (что чревато опечатками) позволяет повысить читаемость, а также хранить все эти числа в одном файле. Поскольку все числа, представляющие коды, записаны в словарь, каждое из них появится всего один раз. Вероятность опечаток при этом значительно снижается.

Преобразование ключей в атрибуты вместо их использования в качестве строк в словаре также снижает риск опечаток. Рассмотрим пример кода из файла `sessions.py`, который гораздо легче прочитать, когда в нем используются слова, а не числа.

```
#
#  ... фрагмент файла sessions.py ...
#      Сокращено с целью показать только самое главное.
#
from .status_codes import codes   ❶


class SessionRedirectMixin(object):   ❷
    def resolve_redirects(self, resp, req, stream=False, timeout=None,
                          verify=True, cert=None, proxies=None,
                          **adapter_kwargs):
        """Получает объект Response.
           Возвращает генератор объектов Response."""

        i = 0
        hist = [] # отслеживаем историю

        while resp.is_redirect:   ❸
            prepared_request = req.copy()

            if i > 0:
                # Обновим историю
                и продолжим отслеживать перенаправления.
                hist.append(resp)
                new_hist = list(hist)
                resp.history = new_hist
            try:
                resp.content  # Освобождаем сокет
            except (ChunkedEncodingError, ContentDecodingError,
                    RuntimeError):
                resp.raw.read(decode_content=False)

            if i >= self.max_redirects:
                raise TooManyRedirects(
                        'Exceeded %s redirects.' % self.max_redirects
                )

            # Вернем соединение в пул.
            resp.close()

            #
            #  ... пропускаем содержимое ...
            #

            # http://tools.ietf.org/html/rfc7231#section-6.4.4
            if (resp.status_code == codes.see_other and   ❹
            method != 'HEAD'):
                method = 'GET'

            # Делаем все, что делают браузеры, несмотря на стандарты...
```

```
# Во-первых, преобразуем коды 302 в запросы GET.
if resp.status_code == codes.found and method != 'HEAD':
    method = 'GET'

# Во-вторых, если в ответ на запрос POST пришел код 301,
# преобразуем его в запрос GET.
# Это странное поведение объяснено в тикете 1704.
if resp.status_code == codes.moved
   and method == 'POST': (5)
    method = 'GET'

#
#  ... и т. д. ...
#
```

(1) Здесь импортируются коды состояний.

(2) Классы-примеси мы опишем в пункте «Примеси (еще одна отличная штука)» подраздела «Примеры структуры из Werkzeug» следующего раздела. Примесь предоставляет методы перенаправления для основного класса `Session`, который определен в том же файле, но не показан в приведенном фрагменте.

(3) Мы входим в цикл, который следует за перенаправлениями и позволяет нам получить желаемое содержимое. Вся логика цикла удалена из этого фрагмента для краткости.

(4) Коды состояний, представленные в виде текста, гораздо читабельнее, нежели числа (которые трудно запомнить): `codes.see_other` в противном случае выглядел бы как `303`.

(5) `codes.found` выглядел бы как `302`, а `codes.moved` — как `301`. Поэтому код становится самозадокументированным; мы можем узнать значение переменных из их имен. Мы избежали засорения кода опечатками, добавив точечную нотацию вместо словаря и строк (например, `codes.found` вместо `codes["found"]`).

Werkzeug

Для того чтобы прочесть код Werkzeug, нам нужно узнать, как веб-серверы общаются с приложениями. В следующих абзацах мы попытались представить максимально короткий обзор по этому вопросу.

Интерфейс Python для взаимодействия серверов с веб-приложениями определен в PEP 333, который написан Филипом Джей Эби (Phillip J. Eby) в 2003 году[1].

[1] С тех пор PEP 333 был заменен спецификацией, в которую были добавлены детали, характерные для Python 3, PEP 3333. Для получения вводной информации рекомендуем прочесть руководство по WSGI (http://pythonpaste.org/do-it-yourself-framework.html) Иэна Бикинга (Ian Bicking).

В нем указано, как веб-сервер (вроде Apache) общается с приложением Python или фреймворком.

1. Сервер будет вызывать приложение каждый раз при получении запроса HTTP (например, GET или POST).
2. Это приложение вернет итерабельный объект, содержащий объекты типа `bytestrings`, который сервер использует для ответа на запрос HTTP.
3. В спецификации также говорится, что приложение примет два параметра, например `webapp(environ, start_response)`. Параметр `environ` будет содержать данные, связанные с запросом, а параметр `start_response` будет функцией или другим вызываемым объектом, который задействуется для отправки обратно заголовка (например, `('Content-type', 'text/plain')`) и состояния (например, `200 OK`) на сервер.

В этом документе есть еще полдюжины страниц дополнительной информации. В середине PEP 333 вы можете увидеть впечатляющее заявление о том, как новый стандарт сделал возможным создание веб-фреймворков.

> Если промежуточное ПО может быть простым и надежным и стандарт WSGI широко доступен в серверах и фреймворках, появляется возможность создания принципиально нового фреймворка Python для веб-приложений, — фреймворка, который состоит из слабо связанных компонентов промежуточного ПО WSGI. Более того, авторы существующих фреймворков могут решить изменить свои сервисы, сделав их похожими на библиотеки, используемые вместе с WSGI, а не на монолитные фреймворки. Это позволит разработчикам приложения выбирать лучшие компоненты для требуемой функциональности, а не довольствоваться одним фреймворком, воспользовавшись всеми его преимуществами и смирившись с недостатками.
>
> Конечно, на текущий момент этот день еще не так близок. В то же время хорошей кратковременной целью для WSGI является возможность использовать любой фреймворк с любым сервером.

Четыре года спустя, в 2007 году, Армин Ронакер (Armin Ronacher) выпустил Werkzeug, намереваясь создать библиотеку WSGI, которую можно использовать для создания приложений WSGI и компонентов промежуточного ПО.

Werkzeug — самый крупный пакет из тех, что мы рассмотрим, поэтому перечислим только несколько его проектных решений.

Читаем код инструментария

Инструментарий (тулкит) — это набор совместимых утилит. В случае Werkzeug все они связаны с приложениями WSGI. Хороший способ понять отдельные утилиты и их предназначение — взглянуть на юнит-тесты, именно так мы и поступим.

Получаем Werkzeug из GitHub:

```
$ git clone https://github.com/pallets/werkzeug.git
$ virtualenv -p python3 venv
$ source venv/bin/activate
(venv)$ cd werkzeug
(venv)$ pip install --editable .
(venv)$ py.test tests  # Запускаем юнит-тесты
```

Читаем документацию Werkzeug

В документации Werkzeug (http://werkzeug.pocoo.org/) перечислены основные возможности, предоставляемые библиотекой, — реализация спецификации WSGI 1.0 (PEP 333, https://www.python.org/dev/peps/pep-0333/), система маршрутизации URL, возможность анализировать и сохранять заголовки HTTP, объекты, которые предоставляют запросы и ответы HTTP, сессии и поддержка cookie, загрузка файлов и другие утилиты и надстройки от сообщества. Плюс полноценный отладчик.

Эти руководства информативны, но мы используем документацию к API вместо того, чтобы изучать компоненты библиотеки. В следующем разделе рассматриваются обертки для Werkzeug (http://werkzeug.pocoo.org/docs/0.11/wrappers/) и документация по маршрутизации (http://werkzeug.pocoo.org/docs/0.11/routing/).

Используем Werkzeug

Werkzeug предоставляет вспомогательные программы для приложений WSGI, поэтому, чтобы узнать, что предоставляет Werkzeug, мы можем начать с приложения WSGI, а затем использовать несколько вспомогательных программ от Werkzeug. Это первое приложение несколько отличается от того, что предоставлено в PEP 333, и не использует Werkzeug. Второе приложение делает то же самое, что и первое, но при этом использует Werkzeug:

```
def wsgi_app(environ, start_response):
    headers = [('Content-type', 'text/plain'), ('charset', 'utf-8')]
    start_response('200 OK', headers)
    yield 'Hello world.'
# Это приложение делает то же самое, что и указанное выше:
response_app = werkzeug.Response('Hello world!')
```

Werkzeug реализует класс `werkzeug.Client`, который ведет себя как реальный веб-сервер при выполнении подобных проверок. Ответ клиента будет иметь тип аргумента `response_wrapper`. В этом коде мы создаем клиентов и используем их для вызова приложений WSGI, которые создали ранее. Для начала разберем простое приложение WSGI (его ответ будет размещен в объекте `werkzeug.Response`):

```
>>> import werkzeug
>>> client = werkzeug.Client(wsgi_app, response_wrapper=werkzeug.Response)
>>> resp=client.get("?answer=42")
```

```
>>> type(resp)
<class 'werkzeug.wrappers.Response'>
>>> resp.status
'200 OK'
>>> resp.content_type
'text/plain'
>>> print(resp.data.decode())
Hello world.
```

Далее используйте объект `werkzeug.Response`:

```
>>> client = werkzeug.Client(response_app, response_wrapper=werkzeug.Response)
>>> resp=client.get("?answer=42")
>>> print(resp.data.decode())
Hello world!
```

Класс `werkzeug.Request` предоставляет содержимое словаря среды (аргумент `environ`, расположенный над `wsgi_app()`) в форме, удобной для пользователя, а также декоратор для преобразования функции, которая принимает объект `werkzeug.Request` и возвращает `werkzeug.Response` приложению WSGI:

```
>>> @werkzeug.Request.application
... def wsgi_app_using_request(request):
...     msg = "A WSGI app with:\n   method: {}\n   path: {}\n   query: {}\n"
...     return werkzeug.Response(
...         msg.format(request.method, request.path, request.query_string))
...
```

Она возвращает следующий код:

```
>>> client = werkzeug.Client(
...     wsgi_app_using_request, response_wrapper=werkzeug.Response)
>>> resp=client.get("?answer=42")
>>> print(resp.data.decode())
A WSGI app with:
   method: GET
   path: /
   query: b'answer=42'
```

Теперь мы знаем, как работать с объектами `werkzeug.Request` и `werkzeug.Response`. Помимо них, в документации указана маршрутизация. Рассмотрим фрагмент кода, где она используется; порядковые номера определяют шаблон и соответствующее ему значение.

```
>>> import werkzeug
>>> from werkzeug.routing import Map, Rule
>>>
>>> url_map = Map([  (1)
...     Rule('/', endpoint='index'),  (2)
...     Rule('/<any("Robin","Galahad","Arthur"):person>',
              endpoint='ask'),  (3)
...     Rule('/<other>', endpoint='other')  (4)
```

```
...     Rule('/<other>', endpoint='other')  ❹
... ])

>>> env = werkzeug.create_environ(path='/shouldnt/match')  ❺
>>> urls = url_map.bind_to_environ(env)
>>> urls.match()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "[...path...]/werkzeug/werkzeug/routing.py", line 1569, in match
    raise NotFound()
werkzeug.exceptions.NotFound: 404: Not Found
```

❶ Объект `werkzeug.Routing.Map` предоставляет основные функции для маршрутизации. Правила соответствия применяются по порядку; первым идет выбранное правило.

❷ Если в строке-заполнителе для правила нет условий в угловых скобках, принимается только полное совпадение, вторым результатом работы метода `urls.match()` является пустой словарь:

```
>>> env = werkzeug.create_environ(path='/')
>>> urls = url_map.bind_to_environ(env)
>>> urls.match()
('index', {})
```

❸ В противном случае второй записью является словарь, в котором соотнесены именованные термы с соответствующими значениями, например `person` соотнесен с `Galahad`:

```
>>> env = werkzeug.create_environ(path='/Galahad?favorite+color')
>>> urls = url_map.bind_to_environ(env)
>>> urls.match()
('ask', {'person': 'Galahad'})
```

❹ Обратите внимание, что `Galahad` мог соответствовать маршруту `other`, но вместо этого ему соответствует `Lancelot`, поскольку выбирается первое соответствующее правило:

```
>>> env = werkzeug.create_environ(path='/Lancelot')
>>> urls = url_map.bind_to_environ(env)
>>> urls.match()
('other', {'other': 'Lancelot'})
```

❺ Если в списке правил соответствие не найдено, генерируется исключение:

```
>>> env = werkzeug.test.create_environ(path='/shouldnt/match')
>>> urls = url_map.bind_to_environ(env)
>>> urls.match()
Traceback (most recent call last):
```

```
File "<stdin>", line 1, in <module>
File "[...path...]/werkzeug/werkzeug/routing.py", line 1569, in match
raise NotFound()
werkzeug.exceptions.NotFound: 404: Not Found
```

Вы соотнесли маршрут запроса с соответствующей конечной точкой. В следующем фрагменте кода продолжим рассматривать суть предыдущего примера:

```
@werkzeug.Request.application
def send_to_endpoint(request):
    urls = url_map.bind_to_environ(request)
    try:
        endpoint, kwargs = urls.match()
        if endpoint == 'index':
            response = werkzeug.Response("You got the index.")
        elif endpoint == 'ask':
            questions = dict(
                Galahad='What is your favorite color?',
                Robin='What is the capital of Assyria?',
                Arthur='What is the air-speed velocity
                        of an unladen swallow?')
            response = werkzeug.Response(questions[kwargs['person']])
        else:
            response = werkzeug.Response("Other: {other}".format(**kwargs))
    except (KeyboardInterrupt, SystemExit):
        raise
    except:
        response = werkzeug.Response(
            'You may not have gone where you intended to go,\n'
            'but I think you have ended up where you needed to be.',
            status=404
        )
    return response
```

Для того чтобы протестировать этот фрагмент, снова используйте класс `werkzeug.Client`:

```
>>> client = werkzeug.Client(send_to_endpoint,
response_wrapper=werkzeug.Response)
>>> print(client.get("/").data.decode())
You got the index.
>>>
>>> print(client.get("Arthur").data.decode())
What is the air-speed velocity of an unladen swallow?
>>>
>>> print(client.get("42").data.decode())
Other: 42
>>>
>>> print(client.get("time/lunchtime").data.decode())  # no match
You may not have gone where you intended to go,
but I think you have ended up where you needed to be.
```

Читаем код Werkzeug

При хорошем тестовом покрытии вы можете узнать, что делает библиотека, взглянув на ее юнит-тесты. Проблема в том, что в этом случае вы смотрите на «отдельные деревья», а не на «лес», то есть исследуете странные варианты использования, предназначенные для того, чтобы гарантировать, что код не даст сбой (вместо того чтобы исследовать связи между модулями). Это годится для инструмента вроде Werkzeug, содержащего модульные, слабо связанные компоненты.

Поскольку мы уже знаем, как работают маршрутизация и обертки для запроса и ответа, то теперь можем прочесть файлы `werkzeug/test_routing.py` и `werkzeug/test_wrappers.py`.

Когда мы в первый раз откроем файл `werkzeug/test_routing.py`, можем быстро взглянуть на связи между модулями, поискав импортированные объекты во всем файле.

Рассмотрим все операторы импорта.

```
import pytest ❶

import uuid ❷

from tests import strict_eq ❸

from werkzeug import routing as r ❹
from werkzeug.wrappers import Response ❺
from werkzeug.datastructures import ImmutableDict, MultiDict ❻
from werkzeug.test import create_environ ❼
```

❶ Конечно, pytest используется для тестирования.

❷ Модуль `uuid` применяется всего в одной функции `test_uuid_converter()`, чтобы подтвердить, что работает преобразование между объектами типа `string` и `uuid.UUID` (строка Universal Unique Identifier (универсальный уникальный идентификатор) позволяет уникально идентифицировать объекты в Интернете).

❸ Функция `strict_eq()` используется довольно часто и определена в файле `werkzeug/tests/__init__.py`. Предназначена для тестирования и нужна только потому, что в Python 2 существовало явное преобразование между строками в формате `Unicode` и `byte`, но на это нельзя полагаться в Python 3.

❹ `werkzeug.routing` — это тестируемый модуль.

❺ Объект `Response` применяется всего в одной функции `test_dispatch()` для подтверждения, что `werkzeug.routing.MapAdapter.dispatch()` передает правильную информацию, отправленную приложению WSGI.

❻ Эти объекты словаря используются по одному разу. `ImmutableDict` нужен для того, чтобы подтвердить, что неизменяемый каталог, указанный в `werkzeug.routing.Map`,

действительно неизменяем, а `MultiDict` — чтобы предоставить несколько значений с ключами строителю URL и подтвердить, что был собран правильный URL.

❼ Функция `create_environ()` предназначена для тестирования: создает среду WSGI без реального запроса HTTP.

Цель этого анализа — исследование связей между модулями. Мы обнаружили, что `werkzeug.routing` лишь импортирует некоторые специальные структуры данных. Остальная часть юнит-тестов показывает область действия модуля маршрутизации. Например, вы можете использовать символы, не входящие в ASCII:

```
def test_environ_nonascii_pathinfo():
    environ = create_environ(u'/лошадь')
    m = r.Map([
        r.Rule(u'/', endpoint='index'),
        r.Rule(u'/лошадь', endpoint='horse')
    ])
    a = m.bind_to_environ(environ)
    strict_eq(a.match(u'/'), ('index', {}))
    strict_eq(a.match(u'/лошадь'), ('horse', {}))
    pytest.raises(r.NotFound, a.match, u'/барсук')
```

Существуют тесты для сборки и анализа URL и даже утилиты для поиска ближайшего совпадения, которое не является полным. Вы можете выполнить пользовательскую обработку в процессе преобразования типов путей и строк URL:

```
def test_converter_with_tuples():
    '''
    Регрессионные тесты для https://github.com/pallets/werkzeug/issues/709
    '''
    class TwoValueConverter(r.BaseConverter):
        def __init__(self, *args, **kwargs):
            super(TwoValueConverter, self).__init__(*args, **kwargs)
            self.regex = r'(\w\w+)/(\w\w+)'
        def to_python(self, two_values):
            one, two = two_values.split('/')
            return one, two
        def to_url(self, values):
            return "%s/%s" % (values[0], values[1])
    map = r.Map([
        r.Rule('/<two:foo>/', endpoint='handler')
    ], converters={'two': TwoValueConverter})
    a = map.bind('example.org', '/')
    route, kwargs = a.match('/qwert/yuiop/')
    assert kwargs['foo'] == ('qwert', 'yuiop')
```

Аналогично немногое импортируется в файле `werkzeug/test_wrappers.py`. В тесте показывается функциональность, доступная объекту Request, — cookies, кодировки, аутентификация, безопасность, таймауты кэша и даже мультиязычные кодировки:

```
def test_modified_url_encoding():
    class ModifiedRequest(wrappers.Request):
        url_charset = 'euc-kr'
    req = ModifiedRequest.from_values(u'/?foo=정상처리'.encode('euc-kr'))
    strict_eq(req.args['foo'], u'정상처리')
```

Как правило, чтение тестов позволяет более подробно ознакомиться с возможностями библиотеки. Теперь понятно, для чего нужна библиотека Werkzeug, поэтому можно двигаться дальше.

Tox в Werkzeug

Tox (https://tox.readthedocs.io/) — инструмент командной строки, который использует виртуальные среды для запуска тестов. Вы можете запускать их на своем компьютере (tox в командной строке), если установлены интерпретаторы Python. Интегрирован с GitHub, поэтому, если у вас есть файл `tox.ini` на высшем уровне репозитория, как у Werkzeug, он автоматически будет запускать тесты при каждом коммите.

Рассмотрим конфигурационный файл Werkzeug `tox.ini` целиком:

```
[tox]
envlist = py{26,27,py,33,34,35}-normal, py{26,27,33,34,35}-uwsgi

[testenv]
passenv = LANG
deps=
# General
    pyopenssl
    greenlet
    pytest
    pytest-xprocess
    redis
    requests
    watchdog
    uwsgi: uwsgi

# Python 2
    py26: python-memcached
    py27: python-memcached
    pypy: python-memcached

# Python 3
    py33: python3-memcached
    py34: python3-memcached
    py35: python3-memcached

whitelist_externals=
    redis-server
    memcached
    uwsgi
```

```
commands=
    normal: py.test []
    uwsgi: uwsgi
           --pyrun {envbindir}/py.test
           --pyargv -kUWSGI --cache2=name=werkzeugtest,items=20 --master
```

Примеры стиля из Werkzeug

В главе 4 мы уже рассмотрели большую часть принятых соглашений по стилю. Первый пример стиля в этом разделе демонстрирует элегантный способ угадать типы на основе строки, второй показывает, что вы можете использовать параметр `VERBOSE` при определении длинных регулярных выражений, поэтому другие пользователи смогут понять, что делает выражение, не затратив на это много времени.

Элегантный способ угадать тип (если реализацию легко объяснить — идея, возможно, хороша)

Вам, скорее всего, приходилось анализировать текстовые файлы и преобразовывать содержимое к разным типам. Это решение выглядит особенно питонским, поэтому мы включили его в книгу.

```
_PYTHON_CONSTANTS = {
    'None':     None,
    'True':     True,
    'False':    False
}


def _pythonize(value):
    if value in _PYTHON_CONSTANTS:  ❶
        return _PYTHON_CONSTANTS[value]
    for convert in int, float:  ❷
        try:  ❸
            return convert(value)
        except ValueError:
            pass
    if value[:1] == value[-1:] and value[0] in '"\'':  ❹
        value = value[1:-1]
return text_type(value)  ❺
```

❶ Поиск в словарях с помощью ключей в Python использует соотнесение хэшей, как и поиск в множестве. Python не имеет операторов `switch case`. (Их отклонили как непопулярные в PEP 3103 (https://www.python.org/dev/peps/pep-3103/).) Вместо этого пользователи Python используют инструкцию `if/elif/else` или (как показано здесь) питонское решение — поиск в словаре.

❷ Обратите внимание, что в первый раз попытка преобразования выполняется к более ограниченному типу `int`, перед тем как попытаться выполнить преобразование к типу `float`.

❸ Питонским решением также является использование оператора `try/except` для `infer type`.

❹ Эта часть необходима, поскольку код находится в файле `werkzeug/routing.py`, а анализируемая строка является частью URL. Здесь проверяется наличие кавычек, при обнаружении они удаляются.

❺ `text_type` преобразовывает строки в формат Unicode таким образом, что они остаются совместимыми и с Python 2, и с Python 3. Этот код практически аналогичен функции `u()`, показанной в разделе «HowDoI» в начале главы 5.

Регулярные выражения (читаемость имеет значение)

Если вы используете в своем коде длинные регулярные выражения, не забывайте про параметр `re.VERBOSE`[1] — сделайте их более понятными для других людей. Пример регулярных выражений показан во фрагменте файла `werkzeug/routing.py`:

```
import re
_rule_re = re.compile(r'''
    (?P<static>[^<]*)                           # static rule data
    <
    (?:
        (?P<converter>[a-zA-Z_][a-zA-Z0-9_]*)   # имя преобразователя
        (?:\((?P<args>.*?)\))?                  # аргументы преобразователя
        \:                                      # разделитель переменных
    )?
    (?P<variable>[a-zA-Z_][a-zA-Z0-9_]*)        # имя переменной
    >
''', re.VERBOSE)
```

Примеры структуры из Werkzeug

В первых двух примерах, связанных со структурой, демонстрируются питонские способы использования динамической типизации. Мы предупреждали, что присваивание переменной разных значений может приводить к появлению проблем (см. подраздел «Динамическая типизация» раздела «Структурируем проект» главы 4), но не упомянули преимущества такого присваивания. Одно из них заключается в том, что вы можете использовать любой тип объекта, который ведет себя предсказуемо. Это называется *утиной типизацией*. Утиная типизация исповедует

[1] re.VERBOSE позволяет писать более читаемые регулярные выражения путем изменения обработки пробелов и добавления комментариев. Подробную информацию вы можете получить из документации к re (https://docs.python.org/3/library/re.html).

следующую философию: «Если что-то выглядит, как утка[1], крякает, как утка, то это и есть утка».

В обоих примерах используется возможность вызвать объекты, которые не являются функциями: вызов `cached_property.__init__()` позволяет проинициализировать экземпляры класса, чтобы их можно быть применять как обычные функции, а вызов `Response.__call__()` позволяет объекту класса `Response` вызвать как функцию самого себя.

В последнем фрагменте используется реализация некоторых классов-примесей (каждый из них определяет какую-либо функциональность в объекте класса `Request`), характерная для Werkzeug, чтобы продемонстрировать, что такие классы — это тоже отличная штука.

Декораторы, основанные на классах (питонский способ использовать динамическую типизацию)

Werkzeug применяет утиную типизацию для того, чтобы создать декоратор `@cached_property`. Когда мы говорили о свойстве, описывая проект `Tablib`, то упоминали, что оно похоже на функцию. Обычно декораторы являются функциями, но поскольку тип ничем не навязывается, они могут быть любым вызываемым объектом: свойство на самом деле является классом. (Вы можете сказать, что оно задумывалось как функция, поскольку его имя не начинается с прописной буквы, а в PEP 8 говорится, что имена классов должны начинаться с прописной буквы.) При использовании нотации, похожей на вызов функции (`property()`), будет вызван метод `property.__init__()` для инициализации и возврата экземпляра свойства — класс, для которого соответствующим образом определен метод `__init__()`, работает как вызываемая функция. Кря.

В следующем фрагменте кода содержится полное определение свойства `cached_property`, которое является подклассом класса `property`. Документация класса `cached_property` говорит сама за себя. Когда это свойство будет использоваться для декорирования `BaseRequest.form` в коде, который мы только что видели, `instance.form` будет иметь тип `cached_property` и с точки зрения пользователя будет вести себя как словарь, поскольку для него определены методы `__get__()` и `__set__()`. При получении доступа к `BaseRequest.form` в первый раз он считает данные формы (если она существует), а затем запишет их в `instance.form.__dict__`, чтобы к ним можно было получить доступ в дальнейшем:

```
class cached_property(property):
    """Декоратор, который преобразует функцию в ленивое свойство.
    Обернутая функция в первый раз вызывается для получения результата,
    затем полученный результат используется при следующем обращении к value::
        class Foo(object):
            @cached_property
```

[1] То есть, если объект можно вызвать, его можно проитерировать или же для него определен правильный метод...

```
            def foo(self):
                # выполняем какие-нибудь важные расчеты
                return 42
    Класс должен иметь '__dict__' для того, чтобы это свойство работало.
    """
    # деталь реализации: для подкласса, встроенного
    # в Python свойства-декоратора
    # мы переопределяем метод __get__ так, чтобы получать кэшированное
    # значение.
    # Если пользователь хочет вызвать метод __get__ вручную, свойство будет
    # работать как обычно, поскольку логика поиска реплицируется
    # в методе __get__ при вызове вручную.
    def __init__(self, func, name=None, doc=None):
        self.__name__ = name or func.__name__
        self.__module__ = func.__module__
        self.__doc__ = doc or func.__doc__
        self.func = func
    def __set__(self, obj, value):
        obj.__dict__[self.__name__] = value
    def __get__(self, obj, type=None):
        if obj is None:
            return self
        value = obj.__dict__.get(self.__name__, _missing)
        if value is _missing:
            value = self.func(obj)
            obj.__dict__[self.__name__] = value
        return value
```

Взглянем на этот код в действии:

```
>>> from werkzeug.utils import cached_property
>>>
>>> class Foo(object):
...     @cached_property
...     def foo(self):
...         print("You have just called Foo.foo()!")
...         return 42
...
>>> bar = Foo()
>>>
>>> bar.foo
You have just called Foo.foo()!
42
>>> bar.foo
42
>>> bar.foo  # Обратите внимание, сообщение не выводится снова...
42
```

Response.__call__

Класс `Response` собран с помощью функциональности класса `BaseResponse`, как и `Request`. Мы изучим его интерфейс и не будем смотреть на сам код. Взглянем

лишь на строку документации для класса `BaseResponse`, чтобы узнать, как его использовать.

```
class BaseResponse(object):

    """Базовый класс ответа. Самая главная особенность, связанная
    с объектом ответа, состоит в том, что это обычное
    WSGI-приложение. Оно инициализируется с помощью нескольких
    параметров ответа (заголовки, тело, код статуса и т. д.)
    и выдает корректный ответ WSGI при вызове с environ,
    а также вызываемую функцию ответа. Поскольку это приложение
    WSGI, обработка обычно заканчивается до того, как реальный
    ответ отправляется на сервер. Это помогает при отладке систем,
    поскольку они могут отлавливать все исключения до того,
    как начинают приходить ответы.
    Рассмотрим небольшой пример приложения WSGI,
    которое используется объектами ответов.

from werkzeug.wrappers import BaseResponse as Response

def index(): (1)
        return Response('Index page')

        def application(environ, start_response): (2)
            path = environ.get('PATH_INFO') or '/'
            if path == '/':
                response = index() (3)
            else:
                response = Response('Not Found', status=404) (4)
            return response(environ, start_response) (5)
"""
    # ... и т. д. ...
```

(1) В примере, показанном в строках документации, функция `index()` вызывается в ответ на запрос HTTP. Ответом будет строка `Index page`.

(2) Эта сигнатура нужна в приложениях WSGI, как указано в PEP 333/PEP 3333.

(3) Класс `Response` является подклассом `BaseResponse`, поэтому ответ представляет собой объект класса `BaseResponse`.

(4) Для ответа 404 требуется лишь установить значение ключевого слова `status`.

(5) И вуаля — объект `response` является вызываемой функцией сам по себе, все сопутствующие заголовки и детали имеют разумные значения по умолчанию (либо переопределены, если путь отличается от `/`).

Как объект класса может быть вызываемой функцией? Дело в том, что для него был определен метод `BaseRequest.__call__`. В следующем примере кода мы покажем лишь этот метод.

```
class BaseResponse(object):
    #
    # ... пропускаем все остальное ...
    #

    def __call__(self, environ, start_response):  (1)
        """Обрабатываем этот ответ как приложение WSGI.
        :param environ: среда WSGI.
        :param start_response: вызываемая функция для ответа,
                               предоставленная сервером WSGI.
        :return: итератор приложения
        """
        app_iter, status, headers = self.get_wsgi_response(environ)
        start_response(status, headers)  (2)
        return app_iter  (3)
```

(1) Эта сигнатура позволяет сделать объекты класса `BaseResponse` вызываемыми функциями.

(2) Здесь учтены требования к вызову приложений WSGI для функции `start_response`.

(3) А здесь возвращается итерабельный объект типа `bytes`.

Пора извлечь следующий урок: если язык позволяет что-то сделать, почему бы не сделать это? После того как мы поняли, что можно добавить метод `__call__()` к любому объекту и сделать его вызываемой функцией, мы можем вернуться к оригинальной документации и еще раз перечитать раздел о модели данных в Python (http://docs.python.org/3/reference/datamodel.html).

Примеси (еще одна отличная штука)

Примеси в Python — это классы, которые предназначены для того, чтобы добавлять определенную функциональность — набор связанных атрибутов. В Python, в отличие от Java, вы можете реализовать множественное наследование. Это означает, что парадигма, при которой создаются подклассы для полдюжины разных классов одновременно, — это один из способов разбить функциональность на отдельные классы. Это похоже на пространства имен.

Подобное разбиение может быть полезно во вспомогательной библиотеке вроде Werkzeug, поскольку она говорит пользователю, какие функции связаны друг с другом, а какие — нет. Разработчик может быть уверен, что атрибуты в одном классе-примеси не будут изменены функциями другого класса-примеси.

В Python для того, чтобы идентифицировать класс-примесь, не нужно ничего, кроме как следовать соглашению, в рамках которого к имени класса добавляется слово «Mixin». Это означает, что, если вы не хотите обращать внимания на порядок разрешения методов, все методы класса-примеси должны иметь разные имена.

В Werkzeug методы класса-примеси иногда могут требовать наличия определенных атрибутов. Эти требования зачастую приводятся в строке документации класса-примеси.

```
# ... из файла werkzeug/wrappers.py

class UserAgentMixin(object):  (1)

    """Добавляет атрибут `user_agent` в объект запроса,
    содержащий проанализированный пользовательский агент
    того браузера, который вызвал запрос как объект
    :class:`~werkzeug.useragents.UserAgent`.
    """

    @cached_property
    def user_agent(self):
        """Текущий агент пользователя."""
        from werkzeug.useragents import UserAgent
        return UserAgent(self.environ)  (2)


class Request(BaseRequest, AcceptMixin, ETagRequestMixin,
              UserAgentMixin, AuthorizationMixin,  (3)
              CommonRequestDescriptorsMixin):

    """Полноценный объект запроса, реализующий следующие примеси:

    - :class:`AcceptMixin` для анализа заголовка
    - :class:`ETagRequestMixin` для работы с etag и управления кэшем
    - :class:`UserAgentMixin` для анализа агента пользователя
    - :class:`AuthorizationMixin` для обработки авторизации
                                  с помощью http
    - :class:`CommonRequestDescriptorsMixin` для обычных заголовков
    """
    (4)
```

(1) В классе `UserAgentMixin` нет ничего особенного; он создает подкласс объекта, что выполняется по умолчанию в Python 3 и настоятельно рекомендуется для совместимости с Python 2. Все это нужно делать явно, поскольку «явное лучше, чем неявное».

(2) `UserAgentMixin.user_agent` предполагает, что существует атрибут `self.environ`.

(3) При включении в список базовых классов для `Request` предоставляемый им атрибут становится доступным с помощью вызова `Request(environ).user_agent`.

(4) Больше ничего нет — мы полностью рассмотрели определение класса `Request`. Вся функциональность предоставляется базовым классом или классами-примесями. Модульными, подключаемыми и такими же великолепными, как Форд Префект.

Новые классы и object

Базовый класс `object` добавляет атрибуты по умолчанию, на которые полагаются другие встроенные параметры. Классы, которые не наследуют от класса `object`, называются старыми или классическими классами. В Python 3 таких классов нет, наследование от класса `object` выполняется по умолчанию. Это означает, что все классы Python 3 являются новыми. Новые классы доступны в Python 2.7 (их поведение не изменялось с версии Python 2.3), но наследование должно быть прописано явно, и мы считаем, что так нужно делать всегда.

Более подробная информация содержится в документации к Python для новых классов (https://www.python.org/doc/newstyle/), в руководстве по адресу http://www.python-course.eu/classes_and_type.php, а история их создания — в статье по адресу http://tinyurl.com/history-new-style-classes. Рассмотрим некоторые различия между старыми и новыми классами (в Python 2.7; все классы Python 3 являются новыми):

```
>>> class A(object):
...     """Новый класс, он является подклассом object."""
...
>>> class B:
...     """Старый класс."""
...
>>> dir(A)
['__class__', '__delattr__', '__dict__', '__doc__', '__format__',
 '__getattribute__', '__hash__', '__init__', '__module__', '__new__',
 '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__',
 '__str__', '__subclasshook__', '__weakref__']
>>>
>>> dir(B)
['__doc__', '__module__']
>>>
>>> type(A)
<type 'type'>
>>> type(B)
<type 'classobj'>
>>>
>>> import sys
>>> sys.getsizeof(A())  # размер в байтах.
64
>>> sys.getsizeof(B())
72
```

Flask

Flask — это веб-микрофреймворк, который объединяет Werkzeug и Jinja2, оба они написаны Армином Роначером (Armin Ronacher). Создавался шутки ради и был выпущен 1 апреля 2010 года, но быстро стал одним из самых популярных

фреймворков Python. Армин несколькими годами ранее (в 2007 году) выпустил Werkzeug, преподнеся его как «швейцарский нож для веб-разработки на Python», и (как мы предполагаем) был немного расстроен тем, что он приживался слишком медленно. Идея Werkzeug заключалась в том, чтобы отвязать WSGI от всего остального, дабы разработчики могли подключать вспомогательные программы по своему выбору. Армин еще не догадывался, насколько нам нужны дополнительные «рельсы»[1].

Читаем код фреймворка

Программный фреймворк похож на физический фреймворк — предлагает структуру для создания приложения WSGI[2]: пользователь библиотеки предоставляет компоненты, которые основное приложение Flask запускает. Наша цель при чтении кода — понять структуру фреймворка и его возможности.

Получаем Flask с GitHub:

```
$ git clone https://github.com/pallets/flask.git
$ virtualenv venv  # вы можете использовать Python 3, но это не рекомендуется
$ source venv/bin/activate
(venv)$ cd flask
(venv)$ pip install --editable .
(venv)$ pip install -r test-requirements.txt  # Required for unit tests
(venv)$ py.test tests  # Запускаем юнит-тесты
```

Читаем документацию Flask

Онлайн-документация к Flask (http://flask.pocoo.org/) начинается с реализации веб-приложения длиной семь строк, затем приводится общее описание Flask: это основанный на Unicode и совместимый с WSGI фреймворк, который использует Jinja2 для создания шаблонов HTML и Werkzeug — для работы со вспомогательными программами WSGI, например маршрутизации URL. Имеет встроенные инструменты для разработки и тестирования.

Вам также доступны полезные руководства, поэтому следующий шаг сделать нетрудно.

---

[1] Отсылка к Ruby on Rails, популяризовавшему веб-фреймворки, следущие стилю Django «все включено», а не стилю Flask «почти ничего не включено (пока вы не добавите надстройки)». Django — отличный выбор в том случае, если вам нужно именно та функциональность, которую предоставляет Django. Он был создан для поддержки онлайн-газеты (и отлично с этим справляется).

[2] WSGI — это стандарт Python, определенный в PEP 333 и PEP 3333, в котором описывается, как приложение может связываться с веб-сервером.

Использование Flask

Мы можем запустить пример `flaskr`, который загрузили с репозитория GitHub. В документах говорится, что это небольшой сайт для ведения блога. Находясь в верхнем каталоге Flask, запустите следующие команды:

```
(venv)$ cd examples/flaskr/
(venv)$ py.test test_flaskr.py  # Тесты должны проходить
(venv)$ export FLASK_APP=flaskr
(venv)$ flask initdb
(venv)$ flask run
```

Читаем код Flask

Итоговой целью Flask является создание веб-приложения, поэтому он не особо отличается от приложений командной строки Diamond и HowDoI. Вместо того чтобы привести еще одну схему, иллюстрирующую прохождение потока выполнения по функциям, мы пройдемся с помощью отладчика по приложению-примеру `flaskr`. Для этого нам нужен pdb — отладчик Python (находится в стандартной библиотеке).

Для начала добавьте точку останова в файле `Flaskr.py`, которая будет активизирована, когда поток выполнения ее достигнет, что заставит интерактивную сессию войти в отладчик:

```
@app.route('/')
def show_entries():
    import pdb; pdb.set_trace()  ## Здесь поставьте точку останова.
    db = get_db()
    cur = db.execute('select title, text from entries order by id desc')
    entries = cur.fetchall()
    return render_template('show_entries.html', entries=entries)
```

Закройте файл и введите `python` в командной строке, дабы войти в интерактивную сессию. Чтобы не запускать сервер, используйте внутренние вспомогательные программы для тестирования Flask — для симуляции запроса HTTP GET к каталогу `/`, куда мы только что поместили отладчик:

```
>>> import flaskr
>>> client = flaskr.app.test_client()
>>> client.get('/')
> /[... truncated path ...]/flask/examples/flaskr/flaskr.py(74)show_entries()
-> db = get_db()
(Pdb)
```

Последние три строки мы получили от `pdb`: мы видим путь (к `Flaskr.py`), номер строки (`74`) и имя метода (`show_entries()`), где мы остановились. Строка (`-> db = get_db()`) показывает выражение, которое будет выполнено следующим, если сделаем в еще один шаг в отладчике. Приглашение (`Pdb`) указывает на то, что мы используем отладчик pdb.

Мы можем подняться или опуститься по стеку[1], введя в командной строке `u` или `d` соответственно. Просмотрите документацию к pdb (https://docs.python.org/library/pdb.html) под заголовком «Команды отладчика», чтобы получить полный список команд, с которыми можно работать. Мы также можем ввести имена переменных, чтобы увидеть их значения, и любую другую команду Python; мы даже можем указать переменным другие значения до того, как продолжим выполнение кода.

Если мы поднимемся по стеку на один шаг, то увидим, что вызвало функцию `show_entries()` (с помощью точки останова, которую мы только что установили): это объект `flask.app.Flask`, имеющий словарь с именем `view_functions`, который соотносит строковые имена (вроде `'show_entries'`) с функциями. Мы также увидим, что функция `show_entries()` была вызвана с `**req.view_args`.

Мы можем узнать, чем является `req.view_args`, в интерактивной командной строке отладчика, просто введя нужное имя (это пустой словарь — `{}`, то есть аргументов нет):

```
(Pdb) u
> /[ ... truncated path ...]/flask/flask/app.py(1610)dispatch_request()
-> return self.view_functions[rule.endpoint](**req.view_args)
(Pdb) type(self)
<class 'flask.app.Flask'>
(Pdb) type(self.view_functions)
<type 'dict'>
(Pdb) self.view_functions
{'add_entry': <function add_entry at 0x108198230>,
'show_entries': <function show_entries at 0x1081981b8>, [... truncated ...]
'login': <function login at 0x1081982a8>}
(Pdb) rule.endpoint
'show_entries'
(Pdb) req.view_args
{}
```

Одновременно с отладкой мы можем открыть файл исходного кода и следовать по нему. Если мы продолжим подниматься по стеку, то увидим, где вызвано приложение WSGI:

```
(Pdb) u
> /[ ... truncated path ...]/flask/flask/app.py(1624)full_dispatch_request()
-> rv = self.dispatch_request()
(Pdb) u
> /[ ... truncated path ...]/flask/flask/app.py(1973)wsgi_app()
-> response = self.full_dispatch_request()
```

[1] В Python стек вызовов содержит исполняемые инструкции, запущенные интерпретатором Python. Поэтому если функция f() вызовет функцию g(), то f() пойдет на стек первой, а g() при вызове будет помещена выше f(). Когда функция g() возвращает значение, она удаляется из стека, и функция f() продолжает работу с того места, где прервалась. Стек называется так потому, что он работает точно так же, как посудомойка, которой требуется вымыть стопку тарелок: новые тарелки помещаются наверх, и вам всегда нужно начинать работу с верхних.

```
(Pdb) u
> /[ ... truncated path ...]/flask/flask/app.py(1985)__call__()
-> return self.wsgi_app(environ, start_response)
```

Если мы продолжим вводить `u`, то окажемся в модуле тестирования, который был использован для создания фальшивого клиента без запуска сервера, — мы достигли конца стека. Мы узнали, что приложение `flaskr` отправляется изнутри объекта класса `flask.app.Flask`, строка 1985 файла `Flask/Flask/app.py`. Перед вами эта функция:

```
class Flask:
    ## ~~ ... пропускаем множество определений ...

    def wsgi_app(self, environ, start_response):
        """Реальное приложение WSGI.
        ... пропускаем остальную документацию ...
        """

        ctx = self.request_context(environ)
        ctx.push()
        error = None
        try:
            try:
                response = self.full_dispatch_request()  (1)
            except Exception as e:
                error = e
                response = self.make_response(self.handle_exception(e))
            return response(environ, start_response)
        finally:
            if self.should_ignore_error(error):
                error = None
            ctx.auto_pop(error)

    def __call__(self, environ, start_response):
        """Shortcut for :attr:`wsgi_app`."""
        return self.wsgi_app(environ, start_response)  (2)
```

(1) Это строка 1973, определенная в отладчике.

(2) Это строка 1985, также определенная в отладчике. Сервер WSGI получит объект Flask как приложение и будет вызывать его всякий раз, когда приходит соответствующий запрос (с помощью отладчика мы нашли точку входа в код).

Мы используем отладчик точно так же, как граф вызовов для HowDoI: следуем вызовам функций (это в принципе не отличается от чтения самого кода). Ценность отладчика заключается в том, что мы не видим лишнего кода, который может отвлечь нас или запутать. Используйте тот подход, который наиболее эффективен для вас.

После того как мы поднялись по стеку с помощью команды `u`, можем вернуться вниз с помощью команды `d` — и снова окажемся у точки останова, помеченной `*** Newest frame`:

```
> /[ ... truncated path ...]/flask/examples/flaskr/flaskr.py(74)show_entries()
-> db = get_db()
(Pdb) d *** Newest frame
```

Далее можем перейти через вызов функции с помощью команды `n` (next — «далее») или же сделать минимально возможный шаг с помощью команды `s` (step — «шаг»):

```
(Pdb) s
--Call--
> /[... truncated path ...]/flask/examples/flaskr/flaskr.py(55)get_db()
-> def get_db():
(Pdb) s
> /[... truncated path ...]/flask/examples/flaskr/flaskr.py(59)get_db()
-> if not hasattr(g, 'sqlite_db'): ❶
##~~
##~~ ... делаем множество шагов и возвращаем
     соединение с базой данных...
##~~
-> return g.sqlite_db
(Pdb) n
> /[... truncated path ...]/flask/examples/flaskr/flaskr.py(75)show_entries()
-> cur = db.execute('select title, text from entries order by id desc')
(Pdb) n
> /[... truncated path ...]/flask/examples/flaskr/flaskr.py(76)show_entries()
-> entries = cur.fetchall()
(Pdb) n
> /[... truncated path ...]/flask/examples/flaskr/flaskr.py(77)show_entries()
-> return render_template('show_entries.html', entries=entries) ❷
(Pdb) n
--Return--
```

Тема на этом не заканчивается, но демонстрировать весь материал довольно утомительно. Вот что мы узнали.

❶ Существует объект `Flask.g`, который при более детальном рассмотрении оказывается глобальным контекстом (при этом он является локальным для объекта Flask). Он предназначен для хранения соединений с базой данных и других устойчивых объектов вроде cookie, которые должны «жить» дольше, чем методы класса Flask. Использование словаря подобным образом позволяет хранить переменные за пределами пространства имен приложения Flask, избегая «столкновений» имен.

❷ Функция `render_template()` находится в конце определения функции в модуле `flaskr.py`. Это означает, что мы, по сути, закончили работу — возвращаемое значение отправляется вызывающей функции из объекта Flask, который мы видели при подъеме по стеку. Поэтому мы пропустим остальную часть.

Отладчик полезно использовать локально в том месте, которое вы проверяете, чтобы точно понять, что происходит в коде до и после выбранной пользователем точки. Одной из основных функций является возможность изменять переменные на ходу (любой код Python работает в отладчике), после чего вы можете продолжить выполнение кода.

Журналирование во Flask

Diamond содержит пример журналирования в приложении, а Flask предоставляет такой пример в библиотеке. Если вы хотите лишь избежать предупреждений «обработчик не обнаружен», выполните поиск строки `logging` в библиотеке Requests (`requests/requests/__init__.py`). Но если вам необходимо предоставить поддержку журналирования в вашей библиотеке или фреймворке, следует воспользоваться примером, предоставляемым Flask.

Журналирование во Flask реализовано в файле `Flask/Flask/logging.py`. В нем определяется формат строк журнала для производства (уровень журналирования `ERROR`) и отладки (уровень журналирования `DEBUG`), также автор кода следует советам из Twelve-Factor App (http://12factor.net/) по записи журналов в потоки (которые направляются в потоки `wsgi.errors` или `sys.stderr` в зависимости от контекста).

Средство журналирования добавляется в основное приложение Flask в `Flask/Flask/app.py` (в следующем фрагменте кода опущена вся нерелевантная информация).

```
# блокировка для инициализации средства журналирования
_logger_lock = Lock()  (1)

class Flask(_PackageBoundObject):

##~~ ... пропускаем другие определения

    #: Имя средства журналирования, которое следует использовать.
    #: По умолчанию используется имя пакета,
    #: переданное в конструктор.
    #:
    #: .. versionadded:: 0.4
    logger_name = ConfigAttribute('LOGGER_NAME') (2)


    def __init__(self, import_name, static_path=None,
                 static_url_path=None,
                 ##~~ ... пропускаем другие аргументы ...
                 root_path=None):
        ##~~ ... пропускаем остальную часть инициализации
        # Подготавливаем отложенную настройку
          для средства журналирования.
        self._logger = None   (3)
        self.logger_name = self.import_name


    @property
    def logger(self):
        """Объект :class:`logging.Logger` для этого приложения.
        Согласно конфигурации по умолчанию журнал записывается
        в поток stderr, если приложение находится в режиме отладки.
```

```
Это средство журналирования может быть использовано
для (сюрприз) записи сообщений в журнал. Рассмотрим
несколько примеров::

    app.logger.debug('A value for debugging')
    app.logger.warning('A warning occurred (%d apples)', 42)
    app.logger.error('An error occurred')

.. versionadded:: 0.3
"""
if self._logger and self._logger.name == self.logger_name:
    return self._logger   ❹
with _logger_lock:   ❺
    if self._logger and self._logger.name == self.logger_name:
        return self._logger
    from flask.logging import create_logger
    self._logger = rv = create_logger(self)
    return rv
```

❶ Данная блокировка используется ближе к концу кода. Блокировки — это объекты, которыми может обладать лишь один поток в любой момент времени. Если объект уже используется, другие потоки, которым нужен этот объект, должны ожидать.

❷ Как и Diamond, Flask использует конфигурационный файл (значения по умолчанию в нем точно такие же, поэтому он здесь не приводится, пользователь может просто ничего не делать и получить полезный ответ), чтобы указать имя средства журналирования.

❸ Средство журналирования для приложения Flask изначально инициализируется как `None`, чтобы его можно было создать позднее (на шаге 5).

❹ Если средство журналирования уже существует, возвращаем его. Декорирование свойств, как показывалось ранее в этой главе, существует для того, чтобы помешать пользователю случайно изменить средство журналирования.

❺ Если средство журналирования еще не существует (оно было инициализировано значением `None`), используем блокировку из шага 1 и создаем его.

Примеры стиля из Flask

Большая часть примеров стиля из главы 4 уже была рассмотрена, поэтому для Flask мы продемонстрируем всего один — реализацию элегантных и простых декораторов маршрутизации.

Декораторы для маршрутизации во Flask (красивое лучше, чем уродливое). Декораторы для маршрутизации во Flask добавляют маршрутизацию URL к целевым функциям, например так:

```
@app.route('/')
def index():
    pass
```

При отправке запроса приложение Flask будет использовать маршрутизацию URL для определения корректной функции, которая нужна для генерации ответа. Синтаксис декоратора позволяет хранить логику маршрутизации за пределами целевой функции, поддерживает функции `flat`, а его использование интуитивно.

Делать это вовсе необязательно — он существует только для того, чтобы предоставить пользователю данную функциональность API. Рассмотрим исходный код метода основного класса Flask, определенного в файле `Flask/Flask/app.py`.

```
class Flask(_PackageBoundObject):  ❶
    """ Объект flask реализует приложение WSGI...
      ... пропускаем остальные строки документации ...
    """
    ##~~ ... пропускаем все, кроме метода routing().

    def route(self, rule, **options):
        """Декоратор, который используется для регистрации функции
        просмотра для заданного правила URL.
        Он аналогичен конструкции :meth:`add_url_rule`,
        но предназначен для использования в декораторах:

            @app.route('/')
            def index():
            return 'Hello World'

        ... пропускаем остальные строки документации ...
        """
        def decorator(f):  ❷
            endpoint = options.pop('endpoint', None)
            self.add_url_rule(rule, endpoint, f, **options)  ❸
            return f
        return decorator
```

❶ `_PackageBoundObject` настраивает файловую структуру для импорта шаблонов HTML, статических файлов и другого содержимого, основываясь на значениях конфигурации, указывающих их местоположение относительно местоположения модуля приложения (например, `app.py`).

❷ Почему бы не назвать его декоратором? Он выполняет те же функции.

❸ Это функция, которая добавляет URL в сопоставление, содержащее все правила. Единственное предназначение `Flask.route` — предоставить удобный декоратор для пользователей библиотеки.

Примеры структуры из Flask

Основной посыл для обоих примеров структуры для Flask — модульность. Flask целенаправленно структурирован так, чтобы вы могли расширять и модифицировать практически все что угодно — от способа кодирования/декодирования строк JSON (Flask дополняет функциональность стандартной библиотеки для работы с JSON возможностью задавать кодировки для объектов datetime и UUID) до классов, использованных для маршрутизации URL.

Стандарты, характерные для приложения (простое лучше, чем сложное)

Flask и Werkzeug имеют модуль `wrappers.py`. Он нужен, чтобы добавить соответствующие значения по умолчанию во Flask, фреймворк для веб-приложений, которые будут дополнять общую библиотеку вспомогательных программ для приложений WSGI от Werkzeug. Во Flask создаются подклассы для `Request` и `Response` Werkzeug для того, чтобы добавлять функциональность, характерную для веб-приложений. Например, объект Response из файла `Flask/Flask/wrappers.py` выглядит так:

```
from werkzeug.wrappers import Request as RequestBase,
Response as ResponseBase
##~~ ... пропускаем все остальное ...

class Response(ResponseBase): ❶
    """Объект ответа, который по умолчанию используется во Flask.
    Он работает так же, как и объект ответа в Werkzeug,
    но по умолчанию имеет mime-тип HTML. Зачастую вам не нужно
    создавать этот объект самостоятельно, поскольку
    :meth:`~flask.Flask.make_response` сделает это за вас.
    Если вы хотите заменить использованный объект ответа, вы можете
    создать подкласс этого объекта и установить значение атрибута
    :attr:`~flask.Flask.response_class` для вашего подкласса. ❷
    """
    default_mimetype = 'text/html'  ❸
```

❶ Класс `Response` от Werkzeug импортируется как `ResponseBase`, эта деталь стиля делает предназначение класса очевидным и позволяет новому подклассу `Response` получить его имя.

❷ Возможность создать подкласс `flask.wrappers.Response`, а также способ выполнения этой задачи подробно описаны в строках документации. Когда вы реализуете подобную функциональность, важно помнить о документации, в противном случае пользователи не будут знать о том, что такая возможность существует.

❸ На этом изменения в классе `Response` заканчиваются. Класс `Request` изменился больше, но мы не будем показывать его в этой главе, чтобы не раздувать ее размер.

В этой небольшой интерактивной сессии показывается, что изменилось в классе `Response`:

```
>>> import werkzeug
>>> import flask
>>>
>>> werkzeug.wrappers.Response.default_mimetype
'text/plain'
>>> flask.wrappers.Response.default_mimetype
'text/html'
>>> r1 = werkzeug.wrappers.Response('hello', mimetype='text/html')
>>> r1.mimetype
u'text/html'
>>> r1.default_mimetype
'text/plain'
>>> r1 = werkzeug.wrappers.Response('hello')
>>> r1.mimetype
'text/plain'
```

Идея изменения mime-типа по умолчанию заключается в том, чтобы позволить пользователям Flask писать меньше кода при сборке объектов ответа, которые содержат HTML (ожидаемые вариант использования Flask). Разумные значения по умолчанию делают ваш код гораздо более понятным для среднестатистического пользователя.

Разумные значения по умолчанию могут быть важны

Иногда значения по умолчанию нужны не только для простоты использования. Например, Flask устанавливает ключи, необходимые для определения числа посетителей и безопасной коммуникации, по умолчанию равными `Null`. Если ключ имеет значение `null`, то приложение при попытке запуска безопасной сессии сгенерирует ошибку. Форсирование появления таких ошибок означает, что пользователи будут создавать собственные тайные ключи — другими (плохими) вариантами будет либо молчаливое разрешение использования ключа сессии, равного `null`, а также небезопасных способов подсчета посетителей, либо предоставление ключа по умолчанию вроде `mysecretkey`, который не будет обновляться (и соответственно, не будет использоваться при развертывании).

Модульность (еще одна отличная штука). Строки документации для `flask.wrappers.Response` сообщают пользователям, что они могут создать подкласс объекта `Response` и использовать его в основном объекте Flask.

В следующем фрагменте кода `Flask/Flask/app.py` мы приведем другие примеры модульности Flask.

```
class Flask(_PackageBoundObject):
    """ ... skip the docstring ...
    """
    #: Класс, который используется для объектов запроса.
```

```
    #: Смотрите :class:`~flask.Request`
    #: для получения более подробной информации.
    request_class = Request  ❶

    #: Класс, который используется для объектов ответа.
    #: Смотрите :class:`~flask.Response` для получения
    #: более подробной информации.
    response_class = Response  ❷

    #: Класс, который используется в среде Jinja.
    #:
    #: .. versionadded:: 0.11
    jinja_environment = Environment  ❸

    ##~~ ... пропускаем другие определения ...

    url_rule_class = Rule
    test_client_class = None
    session_interface = SecureCookieSessionInterface()

    ##~~ .. и т. д. ❹
```

❶ Здесь можно заменить пользовательский класс `Request`.

❷ А здесь вы можете определить пользовательский класс `Response`. Эти атрибуты относятся к классу Flask (а не к объекту), имеют понятные имена, благодаря которым можно узнать их назначение.

❸ Класс `Environment` является подклассом класса `Environment` из Jinja2, который может понимать Flask Blueprints («Эскизы Flask»), что позволяет создавать более крупные приложения, состоящие из множества файлов.

❹ Существуют и другие проявления модульности, которые здесь не показаны, поскольку мы не хотим повторяться.

Если вы взглянете на определение класса Flask, то увидите, где создаются и используются объекты этих классов. Мы показали их для того, чтобы вы поняли, что эти определения классов не должны быть доступны пользователю, — этот выбор был явно сделан при выборе структуры, чтобы дать пользователю больше возможностей контролировать поведение Flask. Когда пользователи говорят о модульности Flask, они имеют в виду не только применение любого бэкенда для базы данных. Подразумевается, что вы можете подключать и применять разные классы.

Вы увидели примеры качественно написанного кода, который следует принципам дзен. Мы рекомендуем взглянуть на полный код перечисленных здесь программ: лучший способ стать хорошим программистом — читать отличный код. И помните: когда писать код становится трудно, используй исходники, Люк![1]

[1] Игра слов use force — use source. — *Примеч. пер.*

6 Отправляем отличный код

В этой главе описываются правила хорошего тона, применяемые при упаковке и отправке кода Python. Вы в основном создаете либо библиотеку Python, которую будут импортировать и использовать другие разработчики, либо отдельное приложение, которое станут применять другие люди, вроде pytest (http://pytest.org/latest/).

Система инструментов, связанная с упаковкой кода Python, стала гораздо более удобной в последние несколько лет благодаря усилиям Python Packaging Authority (PyPA)[1] (https://www.pypa.io/) — это сообщество, которое поддерживает pip, Python Package Index (PyPI), а также большую часть инфраструктуры, связанной с упаковкой в Python. Их документация, касающаяся упаковки (https://packaging.python.org/), отлично составлена, поэтому мы кратко рассмотрим два способа размещения пакетов на частном сайте, а также поговорим о том, как загружать код на Anaconda.org — коммерческий аналог PyPI, запущенный Continuum Analytics.

Недостаток распространения кода с помощью PyPI или другого репозитория пакетов заключается в том, что получатель должен понимать, как инсталлировать требуемую версию Python, и иметь возможность использовать инструменты вроде pip для установки других зависимостей для вашего кода.

Это приемлемо при распространении кода для других разработчиков, но не подходит для распространения приложения пользователям, не являющимся программистами. Для этого вам нужно использовать один из инструментов, показанных в разделе «Замораживаем ваш код».

---

[1] Ходят слухи (https://www.pypa.io/en/latest/history/#before-2013), что они предпочитают называть себя «министерством установки». Ник Коглан (Nick Coghlan), назначенный BDFL для работы с PEP, связанными с упаковкой, написал в своем блоге содержательную статью о самой системе, ее истории и направлении развития (http://bit.ly/incremental-plans).

Тем, кто создает пакеты Python для Linux, нелишне обратить внимание на пакет distro (например, файл с расширением `.deb` на Debian/Ubuntu; он называется дистрибутивом в документации Python). Это потребует выполнения большого объема работы, но мы предложим парочку вариантов в разделе «Упаковка для дистрибутивов, встроенных в Linux». Это похоже на заморозку, но в пакет не входит интерпретатор Python.

Наконец, мы дадим вам отличный совет в разделе «Исполняемые ZIP-файлы»: если ваш код находится в ZIP-архиве (файле с расширением `.zip`) и имеет определенный заголовок, вы можете просто выполнить ZIP-файл. Когда вы знаете, что у вашей целевой аудитории уже установлен Python и ваш проект состоит только из кода Python, этот вариант приемлем.

Использование словаря и Concepts

До появления PyPA не существовало единого и очевидного способа упаковки (вы можете убедиться в этом в исторической дискуссии на Stack Overflow (http://stackoverflow.com/questions/6344076)). В этой главе рассмотрим наиболее важные термины (в глоссарии PyPA (https://packaging.python.org/en/latest/glossary/) вы можете найти и другие определения).

- *Зависимости*. Пакеты Python перечисляют библиотеки, от которых они зависят, в файле `requirements.txt` (для тестирования или развертывания приложения) или в аргументе `install_requires` метода `setuptools.setup()`, когда он вызывается в файле `setup.py`.

  В некоторых проектах могут присутствовать и другие зависимости вроде базы данных Postgres, компилятора C или разделяемого объекта библиотеки C. Они могут быть не указаны явно, но при их отсутствии сборка прервется. Если вы строите библиотеки подобным образом, вам может помочь семинар Пола Керера (Paul Kehrer), посвященный распространению скомпилированных модулей (http://bit.ly/kehrer-seminar).

- *Дистрибутив*. Формат распространения пакета Python (и опционально других ресурсов и метаданных), имеющий форму, которая может быть установлена и затем запущена без дальнейшей компиляции (https://packaging.python.org/en/latest/wheel_egg/).

- *Egg*. Egg (это формат дистрибутива) представляют собой ZIP-файлы с особой структурой, содержащие метаданные для установки. Формат был введен благодаря библиотеке Setuptools и де-факто являлся стандартом многие годы, но никогда не был официальным форматом упаковки в Python. Был заменен на wheels на момент выхода PEP 427. Вы можете прочесть все о различиях между форматами в разделе Wheel vs Egg руководства по упаковке Python.

- *Wheel*. Это формат дистрибутива, который является стандартом для распространения библиотек Python. Они упаковываются как ZIP-файлы с метаданными,

которые pip будет использовать для установки и удаления пакета. Файл имеет расширение .whl и следует соглашению по именованию (указываются платформа, сборка и использованный интерпретатор).

Помимо установленного Python, для запуска обычных пакетов, содержащих лишь код, написанный на Python, не нужно ничего, кроме других библиотек, написанных исключительно на Python, — их можно загрузить с PyPI (https://pypi.python.org/pypi) (или Warehouse (https://warehouse.pypa.io/) — грядущего нового местоположения для PyPI). Сложности (которые мы попытались обойти, добавив дополнительные шаги по установке в главе 2) возникают, когда библиотека Python имеет зависимости за пределами Python, например в библиотеках C или системных исполняемых файлах. Инструменты вроде Buildout и Conda придут на помощь, когда дистрибутив становится сложным даже для формата are Wheel.

Упаковываем код

Упаковка кода дистрибутива означает создание необходимой структуры файла, добавление требуемых файлов и определение подходящих переменных для comform, релевантных PEP, и текущих правил хорошего тона, описанных в разделе Packaging and Distributing Projects («Упаковка и распространение проектов») в Python Packaging Guide[1], или требования к упаковке для других репозиториев вроде http://anaconda.org/.

«Пакет» против «пакета дистрибутива» и против «пакета установки»

Может быть не очень понятно, почему слово *«пакет»* имеет так много значений. В данный момент мы говорим о *пакетах дистрибутива*, которые включают (обычные для Python) пакеты, модули и дополнительные файлы, необходимые для определения релиза. Мы также иногда называем библиотеки *пакетами установки*; они являются высокоуровневыми каталогами пакетов, которые содержат целую библиотеку. Наконец, простой *пакет* — это любой каталог, содержащий файл `__init__.py` и другие модули (файлы с расширением `*.py`). PyPA поддерживает глоссарий терминов, относящихся к упаковке (https://packaging.python.org/en/latest/glossary).

Conda

Если у вас установлен дистрибутив Python от Anaconda, вы все еще можете использовать pip и PyPI, но вашим менеджером пакетов по умолчанию будет `conda`, а репозиторием пакетов — http://anaconda.org/. При сборке пакетов мы рекомендуем

[1] На данный момент существуют два URL, которые ссылаются на одинаковое содержимое: https://python-packaging-user-guide.readthedocs.org/ и https://packaging.python.org.

следовать руководству, расположенному по адресу http://bit.ly/building-conda. Оно заканчивается инструкциями, как загружать пакеты на Anaconda.org.

Если вы создаете библиотеку для научных или статистических приложений — даже если сами не используете Anaconda, — вам понадобится установить дистрибутив для Anaconda, чтобы получить доступ к широкой аудитории научных работников, предпринимателей и пользователей Windows, которые пользуются Anaconda для получения бинарных файлов без лишних забот.

PyPI

Надежная взаимосвязанная система инструментов вроде PyPI и pip позволяет разработчикам максимально легко загружать и устанавливать пакеты как для экспериментов, так и для создания крупных профессиональных систем.

Если вы пишете модуль Python с открытым исходным кодом, PyPI (http://pypi.python.org/), более известный как The Cheeseshop, — походящее место для того, чтобы его разместить[1]. Если для упаковки кода вы выбрали не PyPI, другим разработчикам будет сложнее его найти и использовать как часть их существующих процессов. Они будут относиться к таким проектам с подозрением, поскольку посчитают, что либо проектом плохо управляют и он не готов к релизу, либо проект заброшен.

Правильную и актуальную информацию об упаковке Python вы можете получить из руководства Python Packaging Guide, поддерживаемого PyPA (https://packaging.python.org/en/latest/).

Используйте testPyPI для тестирования и PyPI для реальной разработки

Если вы тестируете настройки упаковки или учите кого-то пользоваться PyPI, можете использовать testPyPI (https://testpypi.python.org/) и запускать свои юнит-тесты до отправки реальной версии в PyPI. Как и в случае PyPI, вы должны изменять номер версии при каждой отправке нового файла.

Пример проекта

Пример проекта PyPA по адресу https://github.com/pypa/sampleproject демонстрирует современные правила хорошего тона, касающиеся упаковки проекта Python. В ком-

[1] В данный момент постепенно осуществляется переход от PyPI к Warehouse (https://warehouse.python.org/) (находится на стадии оценки). Мы можем сказать, что изменился пользовательский интерфейс, но не API. Если вам любопытно, Николь Хэррис (Nicole Harris), один из разработчиков PyPA, написала краткое введение к Warehouse (http://whoisnicoleharris.com/warehouse/).

ментариях к модулю `setup.py` (https://github.com/pypa/sampleproject/blob/master/setup.py) вы найдете полезные советы и описание важных способов управления для PEP. Общая структура файлов организована согласно требованиям, для каждого файла имеются полезные комментарии, повествующие об их предназначении и о том, что должно в них содержаться.

Файл `README` отсылает нас к руководству по упаковке (https://packaging.python.org/) и к руководству по упаковке и распространению (https://packaging.python.org/en/latest/distributing.html).

Используйте pip, но не easy_install

С 2011 года PyPA хорошо поработало для того, чтобы избавиться от заметной путаницы (http://stackoverflow.com/questions/6344076) и прекратить дискуссию (http://stackoverflow.com/questions/3220404) о том, какой именно способ распространения, упаковки и установки библиотек Python можно назвать стандартным. pip был выбран в качестве установщика пакетов по умолчанию в PEP 453 (https://www.python.org/dev/peps/pep-0453/), он устанавливается вместе с Python 3.4 (выпущен в 2014 году) и более поздними версиями[1].

Каждый из этих инструментов может выполнять задачи, которые другим не под силу, а для более старых систем все еще может понадобиться инструмент easy_install. В таблице от PyPA, расположенной по адресу http://packaging.python.org/en/latest/pip_easy_install/, сравниваются pip и easy_install и описываются возможности каждого инструмента.

При разработке собственного кода для установки вам понадобится использовать команду `pip install --editable`, что позволит редактировать код без переустановки.

Личный PyPI

Если вы хотите устанавливать пакеты не только из PyPI (например, из внутреннего рабочего сервера, на котором хранятся проприетарные пакеты компании или пакеты, проверенные и благословленные вашими командами, отвечающими за безопасность и юридические аспекты), можете сделать это, разместив простой сервер HTTP, который работает из каталога, содержащего пакеты, предназначенные для установки.

Предположим, что вы хотите установить пакет, который называется MyPackage.tar.gz. Имеется следующая иерархия каталогов:

```
.
|--- archive/
     |--- MyPackage/
          |--- MyPackage.tar.gz
```

[1] Если у вас установлен Python 3.4 или выше, но нет pip, вы можете установить его из командной строки с помощью команды python —m ensurepip.

Вы можете запустить сервер HTTP из каталога archive, введя следующую команду в оболочке:

```
$ cd archive
$ python3 -m SimpleHTTPServer 9000
```

Это запустит простой сервер HTTP на порту 9000 и перечислит все пакеты (в нашем случае `MyPackage`). Теперь вы можете установить `MyPackage` с помощью любого установщика пакетов для Python. Используйте `pip` в командной строке:

```
$ pip install --extra-index-url=http://127.0.0.1:9000/ MyPackage
```

Очень важно иметь каталог с таким же именем, что и у пакета. Но, если вы чувствуете, что структура MyPackage/MyPackage.tar.gz избыточна, вы всегда можете изъять пакет из каталога и установить его, указав прямой путь:

```
$ pip install http://127.0.0.1:9000/MyPackage.tar.gz
```

Pypiserver

Pypiserver (https://pypi.python.org/pypi/pypiserver) — это минимальный сервер, совместимый с PyPI. Он может использоваться при обслуживании набора пакетов для easy_install или pip. Содержит полезную функциональность вроде административной команды (`-U`), которая обновит все пакеты до самых последних версий, имеющихся в PyPI.

PyPI, размещенный на S3

Помимо вышеперечисленных вариантов, вы можете разместить личный сервер PyPI на Simple Storage Service от Amazon, Amazon S3 (https://aws.amazon.com/s3/). Для этого вам понадобятся учетная запись для Amazon Web Service (AWS) и сегмент S3. Убедитесь, что вы следуете правилам именования сегмента (http://bit.ly/rules-bucket-naming) (вы, конечно, сможете создать сегмент, который нарушает правила именования, но не получите к нему доступ). Для того чтобы использовать ваш сегмент, сначала создайте виртуальную среду на собственной машине и установите требуемые сторонние библиотеки с помощью PyPI или другого источника. Затем установите pip2pi:

```
$ pip install git+https://github.com/wolever/pip2pi.git
```

Далее следуйте указаниям файла README для pip2pi, где вы сможете прочесть о командах `pip2tgz` и `dir2pi`. Вам нужно запустить либо эту команду:

```
$ pip2tgz packages/ YourPackage+
```

либо эти две:

```
$ pip2tgz packages/ -r requirements.txt
$ dir2pi packages/
```

Теперь загружайте свои файлы. Используйте клиент вроде Cyberduck (https://duck.sh/), чтобы каталог с пакетами синхронизировать с вашим сегментом S3. Убедитесь, что вместе с новыми файлами и папками вы загружаете файл `packages/simple/index.html`.

По умолчанию при загрузке новых файлов в сегмент S3 им присваивается уровень доступа «только для пользователей». Если при попытке установить пакет вы получите код HTTP 403, удостоверьтесь, что правильно настроили уровень доступа: используйте веб-консоль от Amazon, чтобы установить уровень `EVERYONE` (разрешает чтение всех файлов). Установите ваш пакет с помощью следующей команды:

```
$ pip install \
  --index-url=http://your-s3-bucket/packages/simple/ \
  YourPackage+
```

Поддержка VCS для pip

Существует возможность получать код непосредственно из системы контроля версий с помощью инструмента pip (для этого следуйте инструкциям по адресу http://bit.ly/vcs-support). Этот способ — альтернатива размещению личного PyPI. Рассмотрим пример команды, использующей pip для получения проекта с GitHub:

```
$ pip install git+git://git.myproject.org/MyProject#egg=MyProject
```

В этой команде `egg` нужно заменить — он носит имя каталога вашего проекта, который вы хотите установить.

Замораживаем код

Фраза «заморозить код» означает «создать отдельный исполняемый пакет, который можно распространять конечным пользователям, на чьих компьютерах не установлен Python». Распространяемый файл или пакет содержит как код приложения, так и интерпретатор Python.

Приложения вроде Dropbox (https://www.dropbox.com/en/help/65), Eve Online (http://www.eveonline.com/), Civilization IV (https://www.civilization.com/en/games/civilization-iv/) и клиента BitTorrent (http://www.bittorrent.com/) (в основном они написаны на Python) делают это.

Преимущество распространения кода таким способом в том, что ваше приложение *будет работать* независимо от того, установлена ли на компьютере пользователя требуемая (или хоть какая-нибудь) версия Python. В Windows и для многих дистрибутивов Linux и OS X корректная версия Python не установлена заранее. Помимо этого, программное обеспечение конечного пользователя всегда должно иметь исполняемый формат. Файлы с расширением `.py` предназначены для программных инженеров и системных администраторов.

Недостаток у заморозки один — она увеличивает размер вашего дистрибутива на 2–12 Мбайт. Кроме того, вам придется отправлять обновленные версии вашего приложения, когда будут выходить обновления безопасности к Python.

Проверяем лицензии при использовании библиотек, написанных на C

Вы должны проверять лицензии для каждого пакета, который используете, на всех уровнях дерева зависимостей. Но мы особенно хотели бы обратить ваше внимание на Windows, поскольку всем решениям для этой операционной системы требуются динамически подключаемые библиотеки (dynamically linked libraries, DLL), написанные на MS Visual C++ и установленные на целевой машине. У вас может не быть разрешения распространять некоторые библиотеки, поэтому вы должны проверять свои возможности перед распространением приложения (см. сообщение Microsoft (http://bit.ly/visual-cplusplus) о файлах Visual C++ для получения более подробной информации). Опционально вы можете использовать компилятор MinGW (https://sourceforge.net/projects/mingw/) (расшифровывается как Minimalist GNU for Windows), но, поскольку это проект GNU, лицензия может быть ограничительной (всегда должна быть открытой и бесплатной).

Кроме того, компиляторы MinGW и Visual C++ не полностью похожи друг на друга, поэтому вы должны проверить, работают ли ваши юнит-тесты так, как вы того ожидаете, после использования другого компилятора. Мы начинаем отходить от основной темы, поэтому проигнорируйте все, что здесь написано, если вы нечасто компилируете код C для Windows, но, например, все еще могут возникнуть кое-какие проблемы при использовании MinGW и NumPy (https://github.com/numpy/numpy/issues/5479). В «Википедии» для NumPy есть статья (https://github.com/numpy/numpy/wiki/Mingw-static-toolchain), в которой рекомендуется применять сборки MinGW со статическими наборами инструментов.

Мы сравним популярные инструменты для заморозки в табл. 6.1. Все они взаимодействуют со стандартной библиотекой Python. Они не могут выполнять кроссплатформенную заморозку[1], поэтому вы должны проводить сборки на целевых платформах.

Инструменты перечислены в том порядке, в каком описаны в этом разделе. PyInstaller и cx_Freeze могут использоваться на всех платформах, py2app работает только для OS X, py2exe — только для Windows, а bbFreeze может работать на UNIX-подобных системах и для Windows, но не для OS X (он еще не портирован на Python 3). Он может генерировать архивы egg, если вам нужна такая функциональность для вашей легаси-системы.

[1] Заморозка кода Python на Linux с его преобразованием для выполнения в ОС Windows применялась в PyInstaller 1.4, но от этой возможности отказались в версии 1.5 (https://github.com/pyinstaller/pyinstaller/wiki/FAQ#features), поскольку она работала только для программ, написанных на чистом Python (то есть не для графических приложений).

Таблица 6.1. Инструменты для заморозки

	pyInstaller	cx_Freeze	py2app	py2exe	bbFreeze
Python 3	Да	Да	Да	Да	—
Лицензия	Модифицированная GPL	Модифицированная PSF	MIT	MIT	Zlib
Windows	Да	Да	—	—	Да
Linux	Да	Да	—	—	Да
OS X	Да	Да	Да	—	—
Eggs	Да	Да	Да	—	Да
Поддержка pkg_resources*	—	—	Да	—	Да
Режим одного файла**	Да	—	—	Да	—

* pkg_resources (https://pythonhosted.org/setuptools/pkg_resources.html) — это отдельный модуль, поставляющийся с Setuptools, который может использоваться для автоматического поиска зависимостей. При заморозке кода возникают трудности, поскольку сложно вручную находить динамически загруженные зависимости статического кода. PyInstaller, например, только говорит о том, что все будет хорошо, когда анализ проводится для egg-архива.

** Режим одного файла — это способ размещения пакетов приложения и всех его зависимостей в едином исполняемом файле в ОС Windows. InnoSetup (http://www.jrsoftware.org/isinfo.php) и Nullsoft Scriptable Install System (NSIS) (http://nsis.sourceforge.net/Main_Page) — это популярные инструменты для создания установщиков, которые могут объединять код в единый файл с расширением .exe.

PyInstaller

PyInstaller (http://www.pyinstaller.org/) может быть использован для создания приложений для OS X, Windows и Linux. Его основное предназначение — быть совместимым со сторонними пакетами после установки, поэтому заморозка сразу работает[1]. По адресу https://github.com/pyinstaller/pyinstaller/wiki/Supported-Packages перечислены пакеты, поддерживаемые PyInstaller. Список поддерживаемых графических библиотек содержит Pillow, pygame, PyOpenGL, PyGTK, PyQT4, PyQT5, PySide (за исключением надстроек для Qt) и wxPython. Поддерживаются также инструменты NumPy, Matplotlib, Pandas и SciPy.

[1] Как мы увидим при взгляде на другие установщики, сложность заключается не в поиске и объединении в пакеты совместимых библиотек, написанных на C, необходимых для определенной версии библиотеки Python, а в обнаружении периферических файлов конфигурации, спрайтов или особой графики, а также других файлов, которые не обнаруживает инструмент для заморозки при исследовании вашего исходного кода.

PyInstaller имеет модифицированную лицензию GPL (https://github.com/pyinstaller/pyinstaller/wiki/License) «с особым исключением, которое позволяет [всем] использовать PyInstaller для сборки и распространения небесплатных программ (включая коммерческие)», поэтому лицензии, которым вы должны подчиняться, будут зависеть от используемых вами при разработке кода библиотек. Команда разработчиков PyInstaller даже предоставляет инструкции, как спрятать исходный код (http://bit.ly/hiding-source-code), для тех, кто создает коммерческие приложения или хочет, чтобы никто не изменял их код. Обязательно прочтите текст лицензии (проконсультируйтесь с адвокатом, если это важно, или воспользуйтесь ресурсом https://tldrlegal.com/, если это неважно), если вам нужно модифицировать их исходный код для того, чтобы создать собственное приложение, поскольку вам, возможно, придется поделиться изменениями.

Руководство к PyInstaller (http://pyinstaller.readthedocs.org/) содержит достаточно информации. Взгляните на страницу http://bit.ly/pyinstaller-reqs, где приводятся требования для PyInstaller, чтобы убедиться, что ваша система совместима: для Windows вам потребуется версия XP или выше, для систем Linux — несколько консольных приложений (списки документации, где вы можете их найти), а для OS X — версия 10.7 (Lion) или выше.

Вы можете использовать Wine (эмулятор Windows) для кросс-компиляции для Windows при работе с Linux или OS X.

Чтобы установить PyInstaller, используйте `pip` из той виртуальной среды, из которой будете строить ваше приложение:

```
$ pip install pyinstaller
```

Для создания стандартного исполняемого файла из модуля `script.py` введите следующую команду:

```
$ pyinstaller script.py
```

Для создания оконного приложения для OS X или Windows задайте флаг `--windowed` в командной строке:

```
$ pyinstaller --windowed script.spec
```

Появляются две новые папки и файл в том же каталоге, где вы выполнили команду `pyinstaller`:

- файл с расширением `.spec`, который можно запустить с помощью PyInstaller для повторного создания сборки;
- каталог сборки, в котором хранятся некоторые файлы журнала;
- каталог `dist`, в котором хранятся главный исполняемый файл и некоторые зависимые библиотеки Python.

PyInstaller помещает все библиотеки Python, используемые вашим приложением, в каталог `dist`, поэтому, когда начнете распространять исполняемый файл, не забудьте, что вы должны распространять весь каталог `dist`.

Файл `script.spec` можно изменить для того, чтобы настроить сборку (http://pythonhosted.org/PyInstaller/#spec-file-operation). Вы можете сделать следующее:

- связать файлы данных и исполняемые файлы;
- включить библиотеки времени выполнения (файлы с расширением `.dll` или `.so`), которые PyInstaller не может определить автоматически;
- добавить для исполняемого файла флаги времени выполнения Python.

Это полезно, поскольку теперь файл можно хранить с помощью системы контроля версий, что упрощает создание будущих сборок. Вики-страница PyInstaller содержит рецепты сборки (https://github.com/pyinstaller/pyinstaller/wiki/Recipes) для некоторых распространенных приложений, включая Django, PyQt4, а также функционал по подписанию кода для Windows и OS X. Там же вы найдете последнюю версию руководства для PyInstaller. Теперь отредактированный файл `script.spec` можно запускать как аргумент для pyinstaller (вместо повторного использования `script.py`):

```
$ pyinstaller script.spec
```

Когда PyInstaller предоставляют файл с расширением .spec, он берет все параметры из содержимого этого файла и игнорирует параметры командной строки, за исключением следующих: --upx-dir=, --distpath=, --workpath=, --noconfirm, и --ascii.

cx_Freeze

Как и PyInstaller, cx_Freeze (https://cx-freeze.readthedocs.org/en/latest/) может замораживать проекты Python для ОС Linux, OS X и Windows. Однако команда разработчиков cx_Freeze не рекомендует компилировать для Windows с помощью Wine, поскольку им пришлось вручную скопировать некоторые файлы, чтобы приложение работало. Для установки спользуйте `pip`:

```
$ pip install cx_Freeze
```

Самый простой способ создать исполняемый файл — запустить `cxfreeze` из командной строки, но есть и другие варианты (вы можете использовать систему контроля версий), если вы пишете сценарий `setup.py`.

Это тот же самый файл `setup.py`, что и для модуля distutils из стандартной библиотеки Python: cx_Freeze расширяет distutils таким образом, чтобы предоставлять несколько дополнительных команд (и модифицировать некоторые другие).

Данные параметры можно передавать через командную строку, сценарий установки или с помощью конфигурационного файла `setup.cfg` (https://docs.python.org/3/distutils/configfile.html).

Сценарий `cxfreeze-quickstart` создает простой файл `setup.py`, который может быть изменен и сохранен в системе контроля версий для будущих сборок. Рассмотрим пример сессии для сценария с именем `hello.py`:

```
$ cxfreeze-quickstart
Project name: hello_world
Version [1.0]:
Description: "This application says hello."
Python file to make executable from: hello.py
Executable file name [hello]:
(C)onsole application, (G)UI application, or (S)ervice [C]:
Save setup script to [setup.py]:
Setup script written to setup.py; run it as:
    python setup.py build
Run this now [n]?
```

Теперь у нас есть сценарий установки и мы можем изменить его в соответствии с нуждами нашего приложения. Параметры вы можете найти в документации к cx_Freeze в разделе *distutils setup scripts* (https://cx-freeze.readthedocs.org/en/latest/distutils.html). Существуют также пример сценария `setup.py` и работающие приложения с минимальной функциональностью, которые показывают, как замораживать приложения, использующие PyQT4, Tkinter, wxPython, Matplotlib, Zope и другие библиотеки. Вы можете найти их в каталоге `samples/` исходного кода cx_Freeze (https://bitbucket.org/anthony_tuininga/cx_freeze/src): перейдите из каталога высшего уровня в `cx_Freeze/cx_Freeze/samples/`. Код также поставляется с установленной библиотекой. Вы можете получить путь, введя следующую команду:

```
$ python -c 'import cx_Freeze; print(cx_Freeze.__path__[0])'
```

Когда закончите редактировать файл `setup.py`, можете использовать его для создания собственного исполняемого файла с помощью одной из этих команд:

```
$ python setup.py build_exe  (1)
$ python setup.py bdist_msi  (2)
$ python setup.py bdist_rpm  (3)
$ python setup.py bdist_mac  (4)
$ python setup.py bdist_dmg  (5)
```

(1) Этот параметр предназначен для создания исполняемого файла для командной строки.

(2) Этот параметр модифицирован cx_Freeze, чтобы вы могли обрабатывать исполняемые файлы Windows и их зависимости.

(3) Этот параметр модифицирован, чтобы вы могли гарантировать, что пакеты для Linux создаются с подходящей архитектурой для текущей платформы.

(4) Этот параметр позволяет создать пакет для отдельного оконного приложения для OS X (.app), содержащий зависимости и исполняемый файл.

(5) Этот параметр позволяет создать пакет app и пакет приложения, которые затем будут упакованы в образ диска DMG.

py2app

py2app (https://pythonhosted.org/py2app) позволяет создать исполняемые файлы для OS X. Как и cx_Freeze, он расширяет distutils, добавляя новую команду py2app. Чтобы установить ее, используйте `pip`:

```
$ pip install py2app
```

Далее автоматически сгенерируйте сценарий `setup.py` с помощью команды `py2applet`:

```
$ py2applet --make-setup hello.py
Wrote setup.py
```

Вы создали простой файл `setup.py`, который можно модифицировать согласно вашим потребностям. Вы можете найти примеры работающих приложений с минимальной функциональностью и соответствующими сценариями `setup.py`, которые используют библиотеки вроде PyObjC, PyOpenGL, pygame, PySide, PyQT, Tkinter и wxPython в исходном коде py2app (https://bitbucket.org/ronaldoussoren/py2app/src/). Для этого перейдите из каталога верхнего уровня в каталог `py2app/examples/`.

Далее запустите файл `setup.py` с помощью команды `py2app`, чтобы создать два каталога — `build` и `dist`. Убедитесь, что вы очистили каталоги перед выполнением повторной сборки. Команда выглядит так:

```
$ rm -rf build dist
$ python setup.py py2app
```

Чтобы прочесть дополнительную документацию, обратитесь к руководству py2app (https://pythonhosted.org/py2app/tutorial.html). Сборка может завершиться генерацией исключения `AttributeError`. Если это произошло, прочтите в руководстве об использовании py2app (http://bit.ly/py2app-tutorial) — запись переменных `scan_code` и `load_module`, возможно, нужно начинать с нижнего подчеркивания: `_scan_code` и `_load_module`.

py2exe

py2exe (https://pypi.python.org/pypi/py2exe) создает исполняемые файлы для Windows. Он очень популярен, клиент BitTorrent (http://www.bittorrent.com/) для Windows был создан с помощью py2exe. Как и cx_Freeze и py2app, он расширяет distutils, в этом случае добавляется команда `py2exe`. Если вам нужно использовать его с Python 2, загрузите более старую версию py2exe из sourceforge (https://sourceforge.net/projects/py2exe/). В противном случае (если у вас установлен Python 3.3+) используйте `pip`:

```
$ pip install py2exe
```

Руководство к py2exe (http://www.py2exe.org/index.cgi/Tutorial) составлено превосходно (возможно, потому что документация размещена на вики-странице, а не в системе контроля версий). Самый простой сценарий `setup.py` выглядит так:

```
from distutils.core import setup
import py2exe
setup(
      windows=[{'script': 'hello.py'}],
)
```

В документации перечислены все параметры конфигурации для py2exe (http://www.py2exe.org/index.cgi/ListOfOptions), а также приведены детальные заметки о том, как (опционально) включить иконки (http://www.py2exe.org/index.cgi/CustomIcons) или создать единый исполняемый файл (http://www.py2exe.org/index.cgi/SingleFileExecutable). В зависимости от лицензии для Microsoft Visual C++ у вас может быть (или не быть) возможность распространять библиотеки среды выполнения для Microsoft Visual C++ вместе с вашим кодом. Если вам это доступно, изучите инструкции по распространению DLL Visual C++ вместе с исполняемыми файлами; в противном случае вы можете предоставить пользователям вашего приложения способ загрузить и установить Microsoft Visual C++ 2008 redistributable packge (http://www.py2exe.org/index.cgi/Tutorial#Step52) или Visual C++ 2010 redistributable packge (http://bit.ly/ms-visual-10), если вы используете Python 3.3 или выше.

После того как вы подкорректируете свой файл установки, можете сгенерировать исполняемый файл в каталог `dist`, введя следующую команду:

```
$ python setup.py py2exe
```

bbFreeze

Библиотеку bbFreeze (https://pypi.python.org/pypi/bbfreeze) в данное время никто не поддерживает, и она не была портирована на Python 3, но ее все еще часто загружают. Как и библиотеки cx_Freeze, py2app и py2exe, она расширяет модуль distutils, добавляя команду `bbfreeze`. Фактически более старые версии `bbFreeze` были основаны на cx_Freeze. Эта библиотека может пригодиться тем, кто поддерживает легаси-системы и хотел бы упаковывать сборки в архивы egg, чтобы затем использовать их в своей инфраструктуре. Для установки используйте команду `pip`:

```
$ pip install bbfreeze  # bbFreeze can't work with Python3
```

Для нее практически нет документации, но вы можете найти алгоритмы сборки (https://github.com/schmir/bbfreeze/blob/master/bbfreeze/recipes.py) для Flup (https://pypi.python.org/pypi/flup), Django, Twisted, Matplotlib, GTK и Tkinter, а также многие другие. Чтобы создать исполняемые бинарные файлы, используйте команду `bdist_bbfreeze` следующим образом:

```
$ bdist_bbfreeze hello.py
```

Она создаст каталог `dist` по адресу, где была запущена `bbfreeze`, в этом каталоге будут находиться интерпретатор Python и исполняемый файл с таким же именем, что и сценарий (в нашем случае `hello.py`).

Для того чтобы сгенерировать архив egg, используйте новую команду `distutils`:

```
$ python setup.py bdist_bbfreeze
```

Существуют и другие варианты, например разметка сборок как снепшотов или ежедневные сборки. Для того чтобы получить больше информации об использовании библиотеки, воспользуйтесь флагом `--help`:

```
$ python setup.py bdist_bbfreeze --help
```

Для более тонкой настройки задействуйте класс `bbfreeze.Freezer` (этот способ использования `bbfreeze` предпочтительный). Он имеет флаги, которые указывают, применять ли сжатие в созданном ZIP-файле, включать ли интерпретатор Python, а также какие сценарии выбрать.

Упаковка дистрибутивов в Linux

Создание дистрибутивов в ОС Linux — это, возможно, «правильный способ» распространения кода для Linux: built distribution похож на замороженный пакет, но не содержит интерпретатор Python, поэтому размер файла меньше примерно на 2 Мбайт по сравнению с замороженным[1]. Если для дистрибутива выходит новое обновление безопасности для Python, ваше приложение автоматически начнет использовать эту версию Python.

Команда `bdist_rpm` из модуля distutils стандартной библиотеки Python позволяет легко создать файл RPM (https://docs.python.org/3/distutils/builtdist.html#creating-rpm-packages) для использования в дистрибутивах Linux вроде Red Hat или SuSE.

Подводные камни при создании пакетов дистрибутивов в Linux

Создание и поддержка разных конфигураций, необходимых для каждого формата дистрибутива (например, `*.deb` для Debian/Ubuntu, `*.rpm` для Red Hat/Fedora, и т. д.), потребует много усилий. Если ваш код — это приложение, которое вы планируете распространять на других платформах, вам также придется создать отдельную конфигурацию, необходимую для заморозки приложения в Windows и OS X. При создании и поддержке одного конфигурационного файла для любого из кроссплатформенных инструментов заморозки, описанных в разделе «Замораживаем код» ранее (это создаст отдельные исполняемые файлы для Windows, OS X и всех дистрибутивов Linux), нужно выполнить куда меньше работы.

[1] Некоторые могли слышать, как их называют бинарными пакетами или установщиками; в Python они имеют официальное имя «дистрибутивы», которое означает RPM, пакеты Debian или исполняемые установщики для Windows. Формат wheel также является разновидностью дистрибутива, но по разным причинам, которые рассматриваются в статье, посвященной wheels, зачастую лучше создавать дистрибутивы для отдельных платформ Linux (http://bit.ly/python-on-wheels), как показано в этом разделе.

> Создание пакета дистрибутива также проблематично, если ваш код предназначен для той версии Python, которая в данный момент не поддерживается дистрибутивом. Вам придется сказать пользователям некоторых версий Ubuntu, что им нужно добавить кое-какой функционал с помощью команд `sudo apt-repository` перед установкой ваших файлов с расширением `.deb` (а это ухудшает мнение о программе). Помимо этого, вам придется поддерживать эквиваленты этих инструкций для каждого дистрибутива и, что еще хуже, обязать пользователей прочитать их и действовать в соответствии с ними.

Обратите внимание на ссылки, которые предоставляют инструкции по упаковке кода Python для некоторых популярных дистрибутивов Linux:

- Fedora (https://fedoraproject.org/wiki/Packaging:Python);
- Debian и Ubuntu (http://bit.ly/debian-and-ubuntu);
- Arch (https://wiki.archlinux.org/index.php/Python_Package_Guidelines).

Если хотите оперативно упаковывать код для всех разновидностей Linux, можете попробовать приложение effing package manager (fpm) (https://github.com/jordansissel/fpm). Оно написано на Ruby и языке оболочки, но нравится нам, поскольку упаковывает код из нескольких источников (включая Python) в файлы для Debian (`.deb`), RedHat (`.rpm`), OS X (`.pkg`), Solaris и других ОС. Оно также позволяет ускорить работу, но не предоставляет дерево зависимостей, поэтому те, кто поддерживает пакет, могут отнестись к нему с неодобрением. Пользователи Debian могут попробовать Alien (http://joeyh.name/code/alien/) — написанную на Perl программу, которая преобразует файлы между форматами Debian, RedHat, Stampede (`.slp`) и Slackware (`.tgz`), но ее код не обновлялся с 2014 года, и те, кто его поддерживал, уже не занимаются этим.

Для тех, кому интересно, Роб МакКуин (Rob McQueen) разместил свои мысли о развертывании серверных приложений при работе в ОС Debian (https://nylas.com/blog/packaging-deploying-python).

Исполняемые ZIP-файлы

Мало кто знает, что Python может выполнять ZIP-файлы, содержащие файл `__main__.py`, начиная с версии 2.6. Это отличный способ упаковки приложений, написанных на чистом Python (приложений, которым не нужны бинарные файлы для указанных платформ). Поэтому, если у вас есть отдельный файл `__main__.py` примерно такого содержания:

```
if __name__ == '__main__':
    try:
```

```
        print 'ping!'
    except SyntaxError:  # Python 3
        print('ping!')
```

и вы создаете ZIP-файл, содержащий его, введите следующую команду:

```
$ zip machine.zip __main__.py
```

Вы сможете отправить этот ZIP-файл другим пользователям; и, если у них установлен Python, они могут выполнить его в командной строке следующим образом:

```
$ python machine.zip
ping!
```

Если хотите создать исполняемый файл, можете добавить в начало ZIP-файла POSIX символы `#!` (называется shebang — «шебанг») — формат ZIP это позволяет, — и у вас получится независимое приложение (если Python доступен по пути, указанном с помощью последовательности символов `#!`). Рассмотрим пример, который продолжает предыдущий код:

```
$ echo '#!/usr/bin/env python' > machine
$ cat machine.zip >> machine
$ chmod u+x machine
```

А теперь его исполняемый файл:

```
$ ./machine
ping!
```

Начиная с Python 3.5 в стандартной библиотеке существует модуль zipapp (https://docs.python.org/3/library/zipapp.html), который позволяет создавать ZIP-файлы более удобным способом. Он также добавляет гибкости, поэтому основной файл не должен обязательно называться __main__.py.

Если вы используете сторонние зависимости, размещая их в текущем каталоге, и изменяете оператор импорта, можете создать исполняемый ZIP-файл, который будет содержать все зависимости. Если структура каталогов выглядит так:

```
.
|--- archive/
     |--- __main__.py
```

и вы работаете в виртуальной среде, где установлены только необходимые вам зависимости, вы можете ввести следующие команды в оболочке, чтобы включить ваши зависимости:

```
$ cd archive
$ pip freeze | xargs pip install --target=packages
$ touch packages/__init__.py
```

Команда `xargs` принимает стандартный вывод от `pip freeze` и превращает его в список аргументов команды `pip`, флаг `--target=packages` отправляет установку в новый каталог `packages`. Команда `touch` создает пустой файл, если таких не существует. В противном случае обновляет временную метку, устанавливая ее значение равным текущему времени. Структура каталога будет выглядеть так:

```
.
|--- archive/
     |--- __main__.py
     |--- packages/
          |--- __init__.py
          |--- dependency_one/
          |--- dependency_two/
```

Убедитесь, что вы изменяете оператор импорта для использования каталога packages, который только что создали:

```
#импортируем dependency_one  # не эту
import packages.dependency_one as dependency_one
```

А затем рекурсивно включите все каталоги в новый ZIP-файл с помощью команды `zip -r`, например так:

```
$ cd archive
$ zip machine.zip -r *
$ echo '#!/usr/bin/env python' > machine
$ cat machine.zip >> machine
$ chmod ug+x machine
```

Часть III. Руководство по сценариям

К этому моменту вы уже установили Python, выбрали редактор, знаете значение слова «питонский», прочитали несколько строк отличного кода Python и можете поделиться собственным кодом с остальным миром. Эта часть руководства поможет вам выбрать библиотеки для вашего продукта независимо от того, что вы решите реализовать. Мы поделимся наиболее распространенными подходами сообщества к определенным сценариям, они сгруппированы по принципу подобия.

- Глава 7 «Взаимодействие с пользователем». Мы рассмотрим библиотеки для всех типов взаимодействия с пользователем — от консольных приложений до графических и веб-приложений.
- Глава 8 «Управление кодом и его улучшение». Мы опишем инструменты для системного администрирования, взаимодействия с библиотеками, написанными на C и C++, а также перечислим способы повысить скорость работы Python.
- Глава 9 «Программные интерфейсы». Мы рассмотрим библиотеки, используемые для работы с сетью, включая асинхронные, а также библиотеки для сериализации и шифрования.
- Глава 10 «Манипуляции с данными». Мы рассмотрим библиотеки, которые предоставляют символьные и численные алгоритмы, графики, а также инструменты для обработки изображений и аудио.
- Глава 11 «Хранение данных». Наконец, изучим отличия между популярными библиотеками для ORM.

7 Взаимодействие с пользователем

Библиотеки, представленные в этой главе, помогают разработчикам писать код для взаимодействия с конечными пользователями. Мы опишем уникальный проект Jupyter и рассмотрим типичные интерфейсы для командной строки и GUI. Завершается глава рассмотрением инструментов для веб-приложений.

Jupyter Notebook

Jupyter (http://jupyter.org/) — это веб-приложение, которое позволяет вам отображать и интерактивно выполнять код Python. Представляет собой интерфейс между пользователями.

Пользователи видят интерфейс клиента Jupyter — написанный на CSS, HTML и JavaScript — в браузере на клиентской машине. Клиент связывается с ядром, написанным на Python (или одном из множества других языков), которое выполняет блоки кода и возвращает результат. Содержимое хранится на серверной машине в формате notebook (`*.nb`) — текст в формате JSON разбит на несколько «клеток», которые могут содержать HTML, Markdown (доступный для прочтения человеком язык разметки; похож на язык, использующийся на вики-страницах), простые заметки или исполняемый код. Сервер может быть локальным (запущен на ноутбуке пользователя) или удаленным, как примеры на сайте https://try.jupyter.org/.

Серверу Jupyter для работы нужна версия Python не ниже 3.3, также он совместим с Python 2.7. Поставляется с наиболее свежими версиями дистрибутивов Python (описаны в разделе «Коммерческие дистрибутивы Python» главы 2), например Canopy и Anaconda, поэтому для этих инструментов не нужно выполнять дополнительную установку, если вы можете компилировать и выполнять сборку

кода C в вашей системе, как мы обсуждали в главе 2. Выполнив подготовку, вы можете установить Jupyter из командной строки с помощью pip:

```
$ pip install jupyter
```

Последние исследования, посвященные использованию Jupyter для обучения (http://bit.ly/jupyter-classroom), показали, что это эффективный и популярный способ создавать интерактивные лекции для учеников, незнакомых с программированием.

Приложения командной строки

Приложения командной строки (также их называют консольными) — это компьютерные программы, предназначенные для использования из текстового интерфейса вроде оболочки (http://en.wikipedia.org/wiki/Shell_(computing)). Они могут быть простыми командами (вроде `pep8` или `virtualenv`) или интерактивными (вроде `pip install`, `pip uninstall` или `pip freeze`) — все они имеют собственные параметры в дополнение к общим параметрам `pip`. Как правило, все они начинают работу из функции `main()`. Наш BDFL поделился мнением (http://bit.ly/python-main-functions) о том, что нужно, чтобы написать хорошее консольное приложение.

Мы используем пример вызова `pip` для перечисления компонентов, которые могут присутствовать в момент вызова приложения командной строки:

```
  ❶   ❷      ❸
$ pip install --user -r requirements.txt
```

❶ *Команда* — это имя вызываемого исполняемого файла.

❷ *Аргументы* следуют за командой и не начинаются с дефиса. Их также называют параметрами и субкомандами.

❸ *Параметры* начинаются либо с одного дефиса (для отдельных символов вроде `-h`), либо с двух (для слов вроде `--help`). Их также называют *флагами* или *переключателями*.

Библиотеки, перечисленные в табл. 7.1, предоставляют разные способы анализа аргументов, а также другие полезные инструменты для приложений командной строки.

Как правило, сначала вы должны использовать инструменты стандартной библиотеки Python. Добавлять другие библиотеки в проект можно только в том случае, если в стандартной библиотеке нет нужной вам функциональности.

В следующих разделах представлена более подробная информация о каждом инструменте, перечисленном в табл. 7.1.

Таблица 7.1. Инструменты для командной строки.

Библиотека	Лицензия	Причины использовать
argparse	Лицензия PSF	• Находится в стандартной библиотеке. • Предоставляет стандартный способ анализа аргументов и параметров
docopt	Лицензия MIT	• Позволяет управлять форматом вспомогательного сообщения. • Анализирует командную строку в соответствии с соглашениями, определенными в стандарте POSIX (http://bit.ly/utility-conventions)
plac	BSD 3-clause	• Автоматически генерирует вспомогательное сообщение на основе существующей сигнатуры функции. • Анализирует аргументы командной строки в фоновом режиме, передавая их непосредственно вашей функции
click	BSD 3-clause	• Предоставляет декораторы для создания вспомогательного сообщения и анализатора (похож на plac). • Позволяет объединять несколько субкоманд. • Взаимодействует с другими надстройками Flask (click не зависит от Flask, но он изначально был написан для того, чтобы помочь пользователям вместе создавать инструменты для командной строки с помощью разных надстроек для Flask, при этом ничего не разрушая, поэтому уже используется в экосистеме Flask)
clint	Лицензия Internet Software Consortium (ISC)	• Имеет такие возможности форматирования, как изменение цвета, добавление отступов и колоночное отображение текста. • Позволяет выполнить проверку типов для данных, введенных интерактивно (например, с помощью регулярного выражения, а также для целого числа или пути). • Дает прямой доступ к списку аргументов, предоставляя простые инструменты для фильтрации и группирования
cliff	Лицензия Apache 2.0	• Предоставляет структурированный фреймворк для крупных проектов Python, имеющих много субкоманд. • Создает интерактивную среду для использования субкоманд без дополнительного кодирования

argparse

Модуль argparse (заменяет устаревший optparse) применяется при анализе параметров командной строки. Интерфейс командной строки, предоставленный проектом HowDoI, использует argparse — вы можете обратиться к нему при создании собственного интерфейса командной строки.

Рассмотрим код генерации анализатора:

```
import argparse
#
# ... пропускаем кучу кода ...
#
def get_parser():
    parser = argparse.ArgumentParser(description='...truncated for brevity...')
    parser.add_argument('query', metavar='QUERY', type=str, nargs='*',
                        help='the question to answer')
    parser.add_argument('-p','--pos',
                        help='select answer in specified position (default: 1)',
                        default=1, type=int)
    parser.add_argument('-a','--all', help='display the full text
                        of the answer',
                        action='store_true')
    parser.add_argument('-l','--link', help='display only the answer link',
                        action='store_true')
    parser.add_argument('-c', '--color', help='enable colorized output',
                        action='store_true')
    parser.add_argument('-n','--num-answers', help='number of answers
                        to return',
                        default=1, type=int)
    parser.add_argument('-C','--clear-cache', help='clear the cache',
                        action='store_true')
    parser.add_argument('-v','--version',
                        help='displays the current version of howdoi',
                        action='store_true')
    return parser
```

Анализатор проверит командную строку и создаст словарь, в котором соотносятся все аргументы и значения. Конструкция `action='store_true'` показывает, что параметр является флагом. При наличии в командной строке он будет сохранен как `True` в словаре анализатора.

docopt

Основной принцип docopt (http://docopt.org/) заключается в том, что документация должна быть красивой и понятной. Библиотека предоставляет одну основную команду `docopt.docopt()`, а также несколько функций и классов для удобства продвинутых пользователей. Функция `docopt.docopt()` принимает инструкции в стиле POSIX, написанные разработчиком, использует их для интерпретации аргументов командной строки и возвращает словарь со всеми аргументами и параметрами, полученными из командной строки. Также она корректно обрабатывает параметры `--help` и `--version`.

В следующем примере значение переменной `arguments` — это словарь, который имеет ключ `name`, `--capitalize` и `--num_repetitions`:

```
#!/usr/bin env python3
"""Здоровается с вами.
  Использование:
    hello <name>... [options]
    hello -h | --help | --version
    -c, --capitalize  писать ли имя с большой буквы
    -n REPS, --num_repetitions=REPS количество повторений [по умолчанию: 1]
"""
__version__ = "1.0.0"  # Необходимо для параметра --version
def hello(name, repetitions=1):
    for rep in range(repetitions):
        print('Hello {}'.format(name))
if __name__ == "__main__":
    from docopt import docopt
    arguments = docopt(__doc__, version=__version__)
    name = ' '.join(arguments['<name>'])
    repetitions = arguments['--num_repetitions']
    if arguments['--capitalize']:
        name = name.upper()
    hello(name, repetitions=repetitions)
```

Начиная с версии 0.6.0 docopt может использоваться для создания сложных программ с субкомандами, которые ведут себя как команды `git` (https://git-scm.com/) или `svn` (https://subversion.apache.org/) (даже если субкоманды написаны на разных языках). Существует полный пример приложения (https://github.com/docopt/docopt/tree/master/examples/git), имитирующий реализацию команды `git`, который показывает, как это возможно.

Plac

Философия Plac (https://pypi.python.org/pypi/plac) заключается в том, что вся информация, необходимая для анализа вызова команды, находится в сигнатуре целевой функции. Библиотека легковесна (примерно 200 строк), оборачивает argparse (http://docs.python.org/2/library/argparse.html) из стандартной библиотеки и предоставляет одну основную команду `plac.plac()`, которая получает анализатор аргумента из сигнатуры функции, анализирует командную строку, а затем вызывает функцию.

Библиотека должна была называться анализатором аргументов командной строки (Command-Line Argument Parser, clap), имя оказалось занято, поэтому она называется Plac — clap, почти наоборот. Руководство по использованию не особо информативно, но посмотрите, как мало строк в этом примере:

```
# hello.py
def hello(name, capitalize=False, repetitions=1):
  """Здоровается с вами."""
    if capitalize:
        name = name.upper()
```

```
    for rep in range(repetitions):
        print('Hello {}'.format(name))
if __name__ == "__main__":
    import plac
    plac.call(hello)
```

Руководство по использованию выглядит так:

```
$ python hello.py --help
usage: hello.py [-h] name [capitalize] [repetitions]
Says hello to you.
positional arguments:
  name
  capitalize   [False]
  repetitions  [1]
optional arguments:
  -h, --help   show this help message and exit
```

Если хотите выполнить преобразование типов для какого-нибудь аргумента до того, как передадите его в функцию, используйте декоратор `annotations`:

```
import plac
@plac.annotations(
    name = plac.Annotation("the name to greet", type=str),
    capitalize = plac.Annotation("use allcaps", kind="flag", type=bool),
    repetitions = plac.Annotation("total repetitions", kind="option",
                  type=int)
def hello(name, capitalize=False, repetitions=1):
  """Здоровается с вами."""
    if capitalize:
        name = name.upper()
    for rep in range(repetitions):
        print('Hello {}'.format(name))
```

Помимо этого, декоратор `plac.Interpreter` предоставляет легковесный способ создать очень быстрое интерактивное приложение для командной строки. Примеры содержатся в документации к интерактивному режиму plac по адресу https://github.com/kennethreitz-archive/plac/blob/master/doc/plac_adv.txt.

Click

Основное предназначение Click (http://click.pocoo.org/) (расшифровывается как Command Line-Interface Creation Kit — набор для создания интерфейсов командной строки) — помочь разработчикам создать компонуемые интерфейсы командной строки, написав минимально возможное количество кода. Документация к Click подтверждает ее связь с docopt.

Функция Click — создавать компонуемые системы, функция docopt — вручную создавать самые красивые интерфейсы командной строки. Эти две цели конфликтуют

друг с другом. Click мешает пользователям реализовывать некоторые шаблоны для того, чтобы сделать интерфейсы командной строки уникальными. Например, вы практически не можете переформатировать вспомогательные страницы.

Стандарты этой библиотеки способны удовлетворить почти все запросы разработчиков, но продвинутые пользователи могут изменить их. Как и в случае с Plac, она задействует декораторы, чтобы привязать определения анализатора к функциям, которые будут их использовать, переместив из самих функций управление аргументами командной строки.

Приложение `hello.py` при использовании Click выглядит так:

```
import click
@click.command()
@click.argument('name', type=str)
@click.option('--capitalize', is_flag=True)
@click.option('--repetitions', default=1,
              help="Times to repeat the greeting.")
def hello(name, capitalize, repetitions):
    """Здоровается, with capitalization и именем."""
    if capitalize:
        name = name.upper()
    for rep in range(repetitions):
        print('Hello {}'.format(name))
if __name__ == '__main__':
    hello()
```

Click анализирует описание из строк документации команды и создает вспомогательное сообщение с помощью пользовательского анализатора, унаследованного от устаревшего члена стандартной библиотеки optparse, который более совместим со стандартами POSIX, чем argparse[1].

Вспомогательное сообщение выглядит так:

```
$ python hello.py --help
Usage: hello.py [OPTIONS] NAME
  Say hello, with capitalization and a name.
Options:
  --capitalize
  --repetitions INTEGER  Times to repeat the greeting.
  --help                 Show this message and exit.
```

Реальная ценность Click заключается в ее модульности — вы можете добавить внешнюю функцию группировки, а затем любую другую функцию, декорированную с помощью click, в ваш проект, и они станут субкомандами для этой команды верхнего уровня:

[1] docopt не использует ни optparse, ни argparse и для анализа строк документации применяет регулярные выражения.

```
import click

@click.group()  ❶
@click.option('--verbose', is_flag=True)
@click.pass_context  ❷
def cli(ctx, verbose):
    ctx.obj = dict(verbose = verbose)  ❸
    if ctx.obj['verbose']:
        click.echo("Now I am verbose.")

#  Функция 'hello' такая же, как раньше...

if __name__ == '__main__':
    cli()  ❹
```

❶ Декоратор `group()` создает команды верхнего уровня, которые запускаются первыми (перед вызванной субкомандой).

❷ С помощью декоратора `pass_context` (опционально) передаются объекты из сгруппированной команды в субкоманду, первый аргумент становится объектом `click.core.Context`.

❸ Этот объект имеет специальный атрибут `ctx.obj`, который можно передавать субкомандам, использующим декоратор `@click.pass_context`.

❹ Теперь вместо функции `hello()` вызывайте функцию, которая была декорирована `@click.group()`; в нашем случае это `cli()`.

Clint

Библиотека Clint (https://pypi.python.org/pypi/clint/) соответствует своему названию и является набором инструментов для работы с интерфейсами командной строки (Command-Line INterface Tools). Поддерживает такую функциональность, как раскрашивание CLI и добавление отступов, индикаторов выполнения, основанных на итераторах, допускает неявную обработку аргументов. Кроме того, это простое и мощное средство отображения столбцов. В этом примере показаны инструменты для раскрашивания и создания отступов:

```
"""Строка использования."""
from clint.arguments import Args
from clint.textui import colored, columns, indent, puts

def hello(name, capitalize, repetitions):
    if capitalize:
        name = name.upper()
    with indent(5, quote=colored.magenta(' ~*~', bold=True)):  ❶
        for i in range(repetitions):
            greeting = 'Hello {}'.format(colored.green(name))  ❷
            puts(greeting)  ❸
```

```
if __name__ == '__main__':
    args = Args()  ❹
    # Сначала выполняем проверку и показываем
    # вспомогательное сообщение
    if len(args.not_flags) == 0 or args.any_contain('-h'):
        puts(colored.red(__doc__))
        import sys
        sys.exit(0)

    name = " ".join(args.grouped['_'].all)  ❺
    capitalize = args.any_contain('-c')
    repetitions = int(args.value_after('--reps') or 1)
    hello(name, capitalize=capitalize, repetitions=repetitions)
```

❶ Использование `indent` интуитивно для менеджера контекста в операторе `with`. Флаг `quote` добавляет в начало каждой строки полужирную фиолетовую конструкцию ~*~.

❷ Модуль `colored` имеет восемь функций для цветов, а также параметр отключения раскрашивания.

❸ Функция `puts()` аналогична `print()`, обрабатывает отступы и кавычки.

❹ `Args` предоставляет простые инструменты фильтрации для списка аргументов. Возвращает еще один объект `Args`, что позволяет объединять фильтры в цепочки.

❺ Использовать аргументы `args`, созданные функцией `Args()`, можно именно так.

cliff

cliff (https://pypi.python.org/pypi/cliff) (Command-Line Interface Formulation Framework — фреймворк для формулирования интерфейсов командной строки) — это фреймворк для создания программ командной строки. Предполагается, что он должен использоваться для создания уникальных мультиуровневых команд, которые ведут себя как `svn` (Subversion) либо `git`, или интерактивных программ вроде оболочек Cassandra или SQL.

Функциональность cliff сгруппирована в абстрактных базовых классах. Вам нужно реализовать `cliff.command.Command` для каждой субкоманды, а затем `cliff.commandmanager.CommandManager` делегирует правильной команде. Рассмотрим минимальную версию программы `hello.py`:

```
import sys

from argparse import ArgumentParser  ❶
from pkg_resources import get_distribution
```

```
from cliff.app import App
from cliff.command import Command
from cliff.commandmanager import CommandManager

__version__ = get_distribution('HelloCliff').version    ❷


class Hello(Command):
    """Здороваемся со всеми."""

    def get_parser(self, prog_name):    ❸
        parser = ArgumentParser(description="Hello command", prog=prog_name)
        parser.add_argument('--num', type=int, default=1, help='repetitions')
        parser.add_argument('--capitalize', action='store_true')
        parser.add_argument('name', help='person\'s name')
        return parser

    def take_action(self, parsed_args): ❹
        if parsed_args.capitalize:
            name = parsed_args.name.upper()
        else:
            name = parsed_args.name
        for i in range(parsed_args.num):
            self.app.stdout.write("Hello from cliff,
            {}.\n".format(name))


class MyApp(cliff.app.App): ❺
    def __init__(self):
        super(MyApp, self).__init__(
            description='Minimal app in Cliff',
            version=__version__,
            command_manager=CommandManager('named_in_setup_py'), ❻
        )


    def main(argv=sys.argv[1:]):
    myapp = MyApp()
    return myapp.run(argv)
```

❶ cliff использует непосредственно `argparse.ArgumentParser` для интерфейса командной строки.

❷ Получаем версию из `setup.py` (при последнем запуске `pip install`).

❸ Для абстрактного базового класса требуется `get_parser()` — он должен возвращать `argparse.ArgumentParser`.

❹ Для абстрактного базового класса требуется `take_action()` — он запускается при вызове команды `Hello`.

❺ В основном приложении создается подкласс `cliff.app.App`, отвечающий за настройку журналирования, потоки ввода/вывода и все остальное, что можно применить ко всем субкомандам.

❻ `CommandManager` управляет всеми классами `Command`, использует содержимое из `entry_points` файла `setup.py` для поиска имен команд.

Приложения с графическим интерфейсом

В этом разделе мы сначала перечислим библиотеки с виджетами — наборы инструментов и фреймворки, которые предоставляют кнопки, полосы прокрутки, индикаторы выполнения и другие заранее созданные компоненты. В конце раздела коснемся игровых библиотек.

Библиотеки виджетов

С точки зрения разработки графических интерфейсов, *виджетами* являются кнопки, слайдеры, полосы прокрутки и другие распространенные элементы интерфейса. С их помощью вам не нужно заниматься низкоуровневым программированием вроде определения того, какая кнопка (если таковая вообще существует) находилась под курсором мыши в момент клика, или работой с разными оконными API, используемыми разными операционными системами.

Если вы никогда не занимались разработкой интерфейсов, то вам нужно что-то, что легко использовать (чтобы вы поняли, как создавать интерфейсы). Мы рекомендуем Tkinter (находится в стандартной библиотеке Python). Вас заинтересуют структура и функции набора инструментов, который лежит в основе библиотеки, поэтому мы сгруппируем библиотеки по тулкитам, начав с самых популярных (табл. 7.2).

Таблица 7.2. Библиотеки виджетов для графических интерфейсов

Библиотека, лежащая в основе (язык)	Библиотека Python	Лицензия	Причины использовать
Tk (Tcl)	tkinter	Лицензия Python Software Foundation	• Все зависимости поставляются вместе с Python. • Предоставляет стандартные виджеты для интерфейса вроде кнопок, полос прокрутки, текстовых окон и полотен
SDL2 (C)	Kivy	MIT или LGPL3 (до версии 1.7.2)	• Может использоваться для создания приложения для Android.

Библиотека, лежащая в основе (язык)	Библиотека Python	Лицензия	Причины использовать
			• Имеет функциональность для работы с технологией мультитач. • Оптимизирована для C там, где это возможно, и использует GPU
Qt (C++)	PyQt	GNU General Public License (GPL) или Commercial	• На всех платформах выглядит одинаково. • Многие приложения и библиотеки уже полагаются на Qt (например, Eric IDE, Spyder и/или Matplotlib), поэтому библиотека может оказаться установленной заранее. • Qt5 (нельзя использовать вместе с Qt4) предоставляет инструменты для создания приложения для Android
Qt (C++)	PySide	GNU Lesser General Public License (LGPL)	Полноценная замена для PyQt, имеющая более либеральную лицензию
GTK (C) (тулкит GIMP)	PyGObject (PyGi)	GNU Lesser General Public License (LGPL)	• Предоставляет связывание с Python для GTK+ 3. • Должна быть знакома всем, кто уже разрабатывал для GNOME
GTK (C)	PyGTK	GNU Lesser General Public License (LGPL)	Применяйте только в том случае, если ваш проект уже использует PyGTK; вы должны портировать старый код PyGTK к PyGObject
wxWindows (C++)	wxPython	Лицензия wxWindows (модифицированная LGPL)	• Предоставляет нативный внешний вид, предлагая различные оконные библиотеки для каждой платформы. • Для разных платформ некоторые фрагменты кода будут отличаться
Objective C	PyObjC	Лицензия MIT	• Предоставляет интерфейс для работы с Objective C. • Придаст вашему проекту для OS X нативный вид. • Не может использоваться на других платформах

В следующих разделах представлена более подробная информация о разных библиотеках для создания графического интерфейса для Python, они сгруппированы по лежащему в их основе тулкиту.

Tk Модуль стандартной библиотеки Tkinter — это тонкий объектно-ориентированный слой, покрывающий Tk, библиотеку виджетов, написанную на языке Tcl. (Вместе они выглядят как Tcl/Tk[1].) Поскольку модуль находится в стандартной библиотеке, он является наиболее удобным и совместимым GUI-тулкитом в нашем списке. Tk и Tkinter доступны на большинстве платформ Unix, а также в Windows и OS X.

На ресурсе TkDocs имеется хорошее руководство по Tk на нескольких языках, содержащее примеры на Python; более подробная информация представлена по адресу http://wiki.python.org/moin/TkInter.

Если у вас есть стандартный дистрибутив Python, у вас должен быть IDLE, графическая интерактивная среда для программирования, написанная исключительно на Python. Она является частью стандартной библиотеки — вы можете запустить ее из командной строки, введя команду `idle`, или просмотреть ее исходный код. Вы можете найти место, где она установлена, введя следующую команду в оболочке:

```
$ python -c"import idlelib; print(idlelib.__path__[0])"
```

В нашем каталоге много файлов, основное приложение IDLE запускается из модуля `PyShell.py`.

Аналогично для того, чтобы просмотреть пример использования интерфейса для рисования, `tkinter.Canvas`, взгляните на код модуля turtle. Вы можете найти его, введя следующую команду в оболочке:

```
$ python -c"import turtle; print(turtle.__file__)"
```

Kivy

Kivy (http://kivy.org/) — это библиотека Python, предназначенная для разработки мультимедиа приложений с поддержкой технологии мультитач. Kivy активно разрабатывается сообществом, имеет разрешительную лицензию в стиле BSD и работает на всех крупных платформах (Linux, OS X, Windows и Android).

Kivy написан на Python и не использует тулкит для работы с окнами, взаимодействует непосредственно с SDL2 (Simple DirectMedia Layer) (https://www.libsdl.org/), библиотекой, написанной на C, которая предоставляет низкоуровневый доступ

[1] Tcl (https://www.tcl.tk/about/language.html) (изначально назывался Tool Command Language — командный язык инструментов) — это легковесный язык, созданный Джоном Аустерхаутом (John Ousterhout) (http://web.stanford.edu/~ouster/cgi-bin/tclHistory.php) в начале 1990-х годов для проектирования интегральных цепей.

к устройствам ввода пользователя[1] и аудио, а также имеет доступ к отрисовке 3D-изображений с помощью OpenGL (или Direct3D для Windows). Имеет несколько виджетов (находятся в модуле `kivy.uix` — https://kivy.org/docs/api-kivy.uix.html), но их не так много, как в наиболее популярных альтернативах вроде Qt и GTK. Если вы разрабатываете традиционное десктопное приложение для бизнеса, больше подойдут Qt или GTK.

Для того чтобы установить библиотеку, перейдите на страницу загрузки Kivy https://kivy.org/#download, найдите свою операционную систему, загрузите ZIP-файл для своей версии Python и следуйте инструкциям для вашей ОС. Помимо кода поставляется и каталог с десятком примеров, которые показывают разные части API.

Qt Qt (произносится «кьют») (http://qt-project.org/) — это кросс-платформенный фреймворк, который широко применяется для разработки графического ПО, но его можно использовать и для разработки приложений без графики. Существует версия Qt5 для Android (http://doc.qt.io/qt-5/android-support.html). Если у вас уже установлен Qt (если вы работаете с Spyder, Eric IDE, Matplotlib или с другим инструментом, использующим Qt), вы можете узнать свою версию Qt из командной строки, введя следующую команду:

```
$ qmake -v
```

Qt выпущен под лицензией LGPL, что позволяет распространять бинарные файлы, которые работают с Qt, если вы не изменяете сам Qt. Коммерческая лицензия позволит воспользоваться такими инструментами, как визуализация данных и покупки в приложении. Qt предоставляет заранее созданные временные платформы для разных типов приложений. Оба интерфейса Python для Qt, PyQt и PySide, не очень хорошо задокументированы, поэтому лучший вариант — воспользоваться документацией к C++ от Qt (http://doc.qt.io/). Кратко опишем каждый интерфейс.

- *PyQt* от компании Riverbank Computing более актуален, нежели PySide (для которого нет версии для Qt5). Для того чтобы установить его, следуйте документации по установке PyQt4 или PyQt5 (http://pyqt.sourceforge.net/Docs/PyQt4/installation.html). PyQt4 работает только с Qt4, а PyQt5 — только с Qt5. (Мы предлагаем воспользоваться Docker — инструментом изоляции, который мы рассматривали в подразделе «Docker» раздела «Инструменты изоляции» главы 3, если вам действительно нужно разрабатывать код с помощью сразу обеих версий, чтобы постоянно не изменять пути к библиотекам.)

[1] В дополнение к поддержке мыши может обрабатывать прикосновения: TUIO (http://www.tuio.org/) (протокол и API с открытым исходным кодом для обработки прикосновений), Wii remote от Nintendo, WM_TOUCH (API, позволяющий работать с прикосновениями, для Windows), USB-тачскрины, использующие продукты HidTouch (https://sourceforge.net/projects/hidtouchsuite/) от Apple, и др. (https://kivy.org/docs/api-kivy.input.providers.html).

Компания Riverbank Computing также публикует pyqtdeploy (https://pypi.python.org/pypi/pyqtdeploy) — графический инструмент, работающий только с PyQt5, который генерирует код C++, характерный для платформы; его вы можете использовать для сборки бинарных файлов дистрибутива. Для получения более подробной информации обратитесь к руководству к PyQt4 (https://pythonspot.com/en/pyqt4/) и к примерам для PyQt5 (https://github.com/baoboa/pyqt5/tree/master/examples).

- *PySide* (https://wiki.qt.io/PySideDocumentation) был выпущен еще тогда, когда Qt принадлежала компании Nokia, поскольку они не могли заставить компанию Riverside Computing, создателей PyQt, изменить лицензию их продукта с GPL на LGPL. Интерфейс предполагается как полноценная замена для PyQt, но несколько замедляет PyQt при разработке. По адресу http://bit.ly/differences-pyside-pyqt описываются различия между PySide и PyQt.

  Для того чтобы установить PySide, следуйте инструкциям из документации к Qt (https://wiki.qt.io/Setting_up_PySide). Существует также страница https://wiki.qt.io/Hello-World-in-PySide, которая поможет вам написать первое приложение с помощью PySide.

GTK+

Библиотека виджетов GTK+ (http://www.gtk.org/) (расшифровывается как GIMP Toolkit — тулкит для GIMP)[1] предоставляет API основной среды рабочего стола GNOME. Программисты могут выбрать GTK+ вместо Qt в том случае, если они предпочитают C и им удобнее смотреть исходный код GTK+ или если они уже разрабатывали приложения GNOME ранее и знакомы с API. Рассмотрим две библиотеки, которые связывают Python и GTK+.

- *pyGTK* предоставляет связывание Python для GTK+, но в данный момент поддерживает только API для GTK+ 2.x (но не для GTK+ 3+). Она больше не поддерживается, и команда разработчиков рекомендует не использовать PyGTK для новых проектов и портировать старые проекты с PyGTK на PyGObject.
- *PyGObject* (известна как PyGI) (https://wiki.gnome.org/Projects/PyGObject) предоставляет связывание с Python, которое позволяет получить доступ ко всей программной платформе GNOME. Библиотека известна как PyGI, поскольку использует (и предоставляет Python API (http://lazka.github.io/pgi-docs/)) для исследования GObject (https://wiki.gnome.org/Projects/GObjectIntrospection), который является API-мостом между другими языками и основными библиотеками GNOME, написанными на C, а также для GLib (https://developer.gnome.org/glib/), если разработчики следуют соглашениям по определению GObject (https://developer.gnome.org/gobject/stable/pt02.html). Библиотека полностью совместима

[1] GIMP расшифровывается как GNU Image Manipulation Program. GTK+ была создана для рисования с GIMP, но стала достаточно популярна для того, чтобы пользователи захотели создать с ее помощью целую оконную среду (так и появилось название GNOME).

с GTK+ 3. Руководство Python GTK+ 3 Tutorial (http://python-gtk-3-tutorial.readthedocs.org/en/latest/) поможет начать работу с библиотекой.

Для того чтобы установить библиотеку, получите бинарные файлы с сайта загрузки PyGObject (http://bit.ly/pygobject-download) или, если пользуетесь OS X, установите ее с помощью homebrew, введя команду `brew install pygobject`.

wxWidgets

Философия, лежащая в основе проекта wxWidgets (https://www.wxwidgets.org/), заключается в том, что лучший способ придать приложению нативный внешний вид — применить нативное API для каждой операционной системы. Qt и GTK+ теперь также используют другие оконные библиотеки помимо X11 за кулисами, но в Qt они абстрактны, а GTK придает им такой внешний вид, будто вы программируете GNOME. Преимущество wXWidgets заключается в том, что вы непосредственно взаимодействуете с каждой платформой, а лицензия позволяет вам даже больше. Проблема, однако, в том, что вам придется работать с каждой платформой по-разному.

Модуль расширения, который оборачивает wxWidgets для пользователей Python, называется wxPython (http://wxpython.org/). Когда-то он был самой популярной оконной библиотекой в Python, возможно, благодаря своей философии использования нативных инструментов для работы с интерфейсом, но теперь обходные решения в Qt и GTK+ стали достаточно удобными. Чтобы установить модуль, перейдите на сайт http://www.wxpython.org/download.php#stable и загрузите соответствующий пакет для вашей ОС. Начните знакомство с руководства для wxPython (http://bit.ly/wxpython-getting-started).

Objective-C

Objective-C — это проприетарный язык, используемый компанией Apple для операционных систем OS X и iOS, также вы можете получить доступ к фреймворку Cocoa для разработки приложений в OS X. В отличие от других вариантов, Objective-C не является кросс-платформенным; он предназначен для продуктов Apple.

PyObjC — это двухсторонний мост между языком Objective-C для OS X и Python, это означает, что не только у Python будет доступ к фреймворку Cocoa для разработки приложений в OS X, но и у программистов Objective-C появится доступ к Python[1].

[1] Создание Swift (http://www.apple.com/swift/) могло снизить потребность в нем — он практически так же прост, как и Python, поэтому, если вы пишете код для OS X, почему бы просто не воспользоваться Swift и сделать все нативно (за исключением вычислений, для которых придется использовать научные библиотеки вроде NumPy и Pandas)?

Фреймворк Cocoa доступен только для OS X, поэтому не выбирайте Objective-C (через PyObjC), если вы пишете кросс-платформенное приложение.

Вам понадобится установить Xcode, как описано в разделе «Установка Python на Mac OS X» в главе 2, поскольку для PyObjC нужен компилятор. PyObjC работает только со стандартным дистрибутивом CPython (но не с дистрибутивами вроде PyPy или Jython), и мы рекомендуем использовать исполняемый файл Python, предоставленный OS X, поскольку эта версия Python была модифицирована компанией Apple и сконфигурирована специально для работы под OS X.

Для того чтобы указать вашей виртуальной среде использовать интерпретатор для Python из вашей системы, применяйте полный путь при его вызове. Если не хотите работать от лица суперпользователя, установите его с помощью переключателя `--user` (это действие сохранит библиотеку в каталоге `$HOME/Library/Python/2.7/lib/python/site-packages/`):

```
$ /usr/bin/python -m pip install --upgrade --user virtualenv
```

Активизируйте среду, войдите в нее и установите PyObjC:

```
$ /usr/bin/python -m virtualenv venv
$ source venv/bin/activate
(venv)$ pip install pyobjc
```

Для этого потребуется какое-то время. PyObjC поставляется вместе с py2app (рассматривается в подразделе «py2app» раздела «Замораживаем код» главы 6), который является инструментом для OS X, позволяющим создавать распространяемые отдельные бинарные файлы приложений. На странице примеров PyObjC (http://pythonhosted.org/pyobjc/examples/index.html) можно найти готовые приложения.

Разработка игр

Kivy очень быстро стал популярным, но он имеет гораздо более крупный отпечаток, нежели библиотеки, перечисленные в этом разделе. Он был указан как фреймворк, поскольку предоставляет виджеты и кнопки, но его часто используют для создания игр. Сообщество Pygame ведет сайт для разработчиков (http://www.pygame.org/hifi.html), применяющих Python, на котором рады всем разработчикам независимо от того, используют они Pygame или нет. Рассмотрим наиболее популярные библиотеки, применяемые для разработки игр.

- *cocos2d* (https://pypi.python.org/pypi/cocos2d). Выпущена под лицензией BSD. Создана на основе pyglet, предоставляя фреймворк для структурирования игры в виде набора сцен, связанных пользовательскими workflows, работой управляет режиссер. Используйте ее, если вам нравится стиль «сцена-режиссер-workflow», который описан в документации по адресу http://tinyurl.com/py-cocos2d-scenes, или если хотите задействовать pyglet для рисования и SDL2 для джойстика и аудио.

Вы можете установить cocos2D с помощью pip. Что касается SDL2, сначала проверьте его наличие в вашем менеджере пакетов, а затем загрузите с сайта https://www.libsdl.org/. Лучший способ начать работу — изучить поставляемые примеры приложений для cocos2d (https://github.com/los-cocos/cocos/tree/master/samples).

- *pyglet* (https://pypi.python.org/pypi/pyglet). Выпущена под лицензией BSD. Представляет собой набор легковесных оболочек для OpenGL, а также инструменты для представления и перемещения спрайтов по окну. Просто установите ее — вам должно хватить pip, поскольку практически на каждом компьютере имеется OpenGL — и запустите несколько примеров приложений (https://bitbucket.org/pyglet/pyglet/src/default/examples), которые включают в себя полноценный клон игры Asteroids (http://bit.ly/astrea-py) (для его написания потребовалось менее чем 800 строк кода).
- *Pygame*. Выпущена под лицензией Zlib, а также под лицензией GNU LGPLv2.1 для SDL2. Существует крупное активное сообщество, вступив в которое, вы можете получить множество руководств по Pygame (http://www.pygame.org/wiki/tutorials), но члены этого сообщества используют SDL1, предыдущую версию библиотеки. Она недоступна в PyPI, поэтому сначала поищите ее в своем менеджере пакетов, а затем, если ее нет, загрузите Pygame (http://www.pygame.org/download.shtml).
- *Pygame-SDL2* (http://bit.ly/pygame-sdl2) была недавно объявлена как попытка заново реализовать Pygame с бэкендом в виде SDL2. Выпущена под теми же лицензиями, что и Pygame.
- *PySDL2* (https://pypi.python.org/pypi/PySDL2) работает на CPython, IronPython и PyPy, является тонким интерфейсом к библиотеке SDL2. Если вам нужен самый маленький интерфейс между SDL2 и Python, эта библиотека подойдет идеально. Для получения более подробной информации смотрите руководство к PySDL2 (http://pysdl2.readthedocs.io/en/latest/tutorial/index.html).

Веб-приложения

Поскольку мощный язык сценариев был адаптирован как для быстрого прототипирования, так и для больших проектов, Python широко используется в разработке веб-приложений (на Python написаны YouTube, Pinterest, Dropbox и The Onion).

Две библиотеки, которые мы рассмотрели в главе 5 — Werkzeug и Flask, — были связаны со сборкой веб-приложений. С их помощью мы кратко описали интерфейс Web Server Gateway Interface (WSGI), стандарт Python, определенный в PEP 3333, который указывает, как общаются веб-серверы и веб-приложения, написанные на Python.

В этом разделе мы рассмотрим веб-фреймворки, написанные на Python, их систему шаблонов, серверы, с которыми они взаимодействуют, а также платформы, на которых они запускаются.

Веб-фреймворки/микрофреймворки

Веб-фреймворк состоит из набора библиотек и основного обработчика, внутри которого вы можете строить собственный код для реализации веб-приложения (то есть интерактивного сайта, предоставляющего его клиенту интерфейс к коду, запущенному на сервере). Большая часть веб-фреймворков включает в себя шаблоны и вспомогательные программы, позволяющие выполнить как минимум следующий действия.

- *Маршрутизация URL*. Соотнесение входящего запроса HTTP с соответствующей функцией Python (или вызываемой функцией).
- *Обработка объектов Request и Response*. Инкапсуляция информации, полученной или отправленной браузеру пользователя.
- *Шаблоны*. Внедрение переменных Python в шаблоны HTML или другую форму вывода информации, что позволяет программистам отделить логику приложения (в Python) от макета (в шаблоне).
- *Веб-сервис для отладки*. Запуск миниатюрного сервера HTTP на машинах разработчиков, позволяющего ускорить разработку; зачастую код на серверной стороне автоматически перезагружается при обновлении файлов.

Вам не нужно писать код вокруг фреймворка. Он изначально должен предоставлять все, что вам требуется, при этом функциональность должна быть протестирована другими разработчиками, поэтому, если вы все еще не можете найти то, что вам нужно, продолжайте исследовать другие доступные фреймворки (например, Bottle, Web2Py, CherryPy). Технический редактор также отметил, что мы должны упомянуть Falcon (http://falconframework.org/) — фреймворк, предназначенный для сборки RESTful API (но не для работы с HTML).

Все библиотеки, перечисленные в табл. 7.3, могут быть установлены с помощью команды `pip`:

```
$ pip install Django
$ pip install Flask
$ pip install tornado
$ pip install pyramid
```

Таблица 7.3. Веб-фреймворки

Библиотека Python	Лицензия	Причины использовать
Django	Лицензия BSD	• Предоставляет структуру — в основном заранее созданный сайт, где вы разрабатываете макет и данные/логику. • Автоматически генерирует административный веб-интерфейс, где непрограммисты могут добавлять или удалять данные (например, новостные статьи).

Библиотека Python	Лицензия	Причины использовать
		• Поставляется вместе с инструментом для объектно-реляционного отображения (object-relational mapping, ORM)
Flask	Лицензия BSD	• Позволяет полностью контролировать все, что находится у вас в стеке. • Предоставляет элегантные декораторы, которые добавляют маршрутизацию URL для любой выбранной вами функции. • Позволяет вам отказаться от структуры, предоставленной Django или Pyramid
Tornado	Лицензия Apache 2.0	• Предоставляет отличную функциональность по обработке событий — Tornado использует собственный сервер HTTP. • Предоставляет способ обработать множество WebSockets (полнодуплексная, устойчивая коммуникация с помощью TCP*) или других долгоиграющих соединений сразу после установки
Pyramid	Модифицированная лицензия BSD	• Предоставляет заранее собранную структуру, которая называется временной платформой, но в меньшей степени, чем Django, позволяет использовать любой интерфейс базы данных или библиотеку шаблонов. • Основан на популярном фреймворке Zope и на Pylons

* Transmission Control Protocol (TCP) — стандартный протокол, который определяет способ, с помощью которого два компьютера устанавливают соединение друг с другом.

В следующих разделах приведена более подробная информация о веб-фреймворках из табл. 7.3.

Django

Django (http://www.djangoproject.com/) — это готовый к работе веб-фреймворк. Отличный выбор для тех, кто создает сайты, ориентированные на содержимое. Предоставляя множество вспомогательных программ и шаблонов сразу после установки, Django позволяет создавать сложные веб-приложения, работающие с базами данных.

У Django имеется крупное и активное сообщество. Многие их модули, которые можно использовать повторно (http://djangopackages.com/), вы можете встроить в свой проект или настроить так, как вам нужно.

Проводятся ежегодные конференции в Соединенных Штатах (http://djangocon.us/) и Европе (http://djangocon.eu/), посвященные Django, — большая часть веб-приложений Python сегодня создается с помощью Django.

Flask

Flask (http://flask.pocoo.org/) — это микрофреймворк для Python. Отлично подойдет для сборки небольших приложений, API и веб-сервисов. Вместо предоставления всех инструментов, которые теоретически могут понадобиться, во Flask реализованы самые распространенные компоненты фреймворка для создания веб-приложений вроде маршрутизации URL, объекты запросов и ответов HTTP, а также шаблоны. Создание приложения с помощью Flask похоже на написание стандартного модуля Python, только к некоторым функциям прикреплены маршруты (с применением декоратора, как это показано в следующем фрагменте кода). Выглядит красиво:

```
@app.route('/deep-thought')
def answer_the_question():
    return 'The answer is 42.'
```

Если вы используете Flask, ответственность за выбор других компонентов приложения (если таковые нужны) ложится на вас. Например, доступ к базам данных или генерация/проверка форм не встроены во Flask (поскольку многим веб-разработчикам эта функциональность не нужна). Если вам это требуется, существует множество доступных расширений (http://flask.pocoo.org/extensions/) вроде SQLAlchemy (http://flask-sqlalchemy.pocoo.org/) для баз данных, pyMongo (https://docs.mongodb.org/getting-started/python/) для MongoDB и WTForms (https://flask-wtf.readthedocs.org/) для работы с формами.

Flask является выбором по умолчанию для любого веб-приложения, которое не подходит для заранее созданных временных платформ от Django. Попробуйте запустить эти примеры приложений для Flask (https://github.com/pallets/flask/tree/master/examples), чтобы ознакомиться с фреймворком. Если хотите запустить несколько приложений (по умолчанию для Django), применяйте инструмент для планирования (http://bit.ly/application-dispatching). Если желаете дублировать наборы подстраниц внутри приложения, используйте Flask's Blueprints («Эскизы Flask») (http://flask.pocoo.org/docs/0.10/blueprints/).

Tornado

Tornado (http://www.tornadoweb.org/) — это асинхронный (управляемый событиями и неблокирующий, как Node.js (https://nodejs.org/en/)) веб-фреймворк для Python, который имеет собственный цикл событий[1]. Это позволяет нативно поддерживать, например, протокол коммуникации WebSockets (http://bit.ly/websockets-api). В отличие от других фреймворков, показанных в этом разделе, Tornado не является при-

[1] Он был вдохновлен веб-сервером проекта Twisted (http://twistedmatrix.com/trac/wiki/TwistedWeb), который является частью тулкита Tornado для работы с сетями. Если вы хотите воспользоваться функциональностью, которой нет в Tornado, попробуйте поискать ее в Twisted. Но предупреждаем: новичкам трудно освоить Twisted.

ложением WSGI (http://www.tornadoweb.org/en/stable/wsgi.html). Его можно запустить как приложение или сервер WSGI с помощью модуля `tornado.wsgi`, но даже его авторы спросят: «А в чем смысл?»[1], поскольку WSGI — это *синхронный* интерфейс, а Tornado — *асинхронный* фреймворк.

Tornado сложнее, чем Django или Flask, и используется гораздо реже. Работайте с ним только в том случае, если прирост производительности, связанный с применением асинхронного фреймворка, будет стоить дополнительного времени, затраченного на программирование. Если решитесь, хорошей стартовой точкой станут демонстрационные приложения (https://github.com/tornadoweb/tornado/tree/master/demos). Качественно написанные приложения Tornado славятся отличной производительностью.

Pyramid

Pyramid (http://www.pylonsproject.org/) похож на Django, однако больший упор в нем делается на модульность. Поставляется вместе с несколькими встроенными библиотеками (меньшее количество функциональности доступно сразу) и поощряет расширение его базовой функциональности с помощью шаблонов, которые называются временными платформами.

Вы регистрируете временную платформу, а затем вызываете ее при создании нового проекта с помощью команды `pcreate` — аналогично команде Django `django-admin startproject project-name command`, но у вас имеются параметры для работы с разными структурами, бэкендами для баз данных и маршрутизацией URL.

Pyramid не очень востребован у пользователей, в отличие от Django и Flask. Это хороший фреймворк, но он не считается популярным выбором для создания новых веб-приложений на Python.

По адресу http://docs.pylonsproject.org/projects/pyramid-tutorials можно найти несколько руководств для Pyramid. Чтобы убедить вашего босса использовать Pyramid, взгляните на портал https://trypyramid.com/, где имеется большой объем информации о Pyramid.

Движки для веб-шаблонов

Большая часть приложений WSGI предназначена для реагирования на запросы HTTP и обслуживания содержимого в формате HTML или других языках разметки. Движки шаблонов отвечают за отрисовку этого содержимого: управляют набором файлов шаблонов, имеют систему иерархии и включения, позволяющую избежать лишнего повторения, и заполняют статическое содержимое шаблонов

[1] На самом деле в документации относительно WSGI сказано следующее: «Используйте WSGIContainer только в том случае, когда преимущества объединения Tornado и WSGI в одном процессе перекрывают сниженную масштабируемость».

динамическим содержимым, сгенерированным приложением. Это помогает придерживаться *концепции разделения обязанностей*[1] — мы размещаем логику в коде, а отрисовку доверяем шаблонам.

Файлы шаблонов иногда пишутся дизайнерами или фронтенд-разработчиками, и сложность страниц может усложнить координирование. Рассмотрим правила хорошего тона как для приложений, передающих динамическое содержимое движку, так и для самих шаблонов.

- *Никогда зачастую лучше, чем прямо сейчас.* Файлам шаблонов должно передаваться только то динамическое содержимое, которое нужно отрисовать. Постарайтесь не передавать дополнительное содержимое на всякий случай: гораздо проще добавить отсутствующие переменные, чем убрать переменную, которая, скорее всего, не будет использоваться в работе.
- *Постарайтесь держать логику за пределами шаблона.* Многие движки шаблонов позволяют создавать сложные выражения или операции присваивания в самом шаблоне, а также встраивать код Python, который будет оценен именно в шаблоне. Это может привести к неконтролируемому росту сложности и зачастую усложняет поиск ошибок. Мы не против такого подхода — практичность побеждает красоту, — просто держите себя в руках.
- *Отделяйте JavaScript от HTML.* Зачастую необходимо смешивать шаблоны JavaScript с шаблонами HTML. Не теряйте голову: изолируйте те части, где шаблон HTML передает переменные в код JavaScript.

Все движки шаблонов, перечисленные в табл. 7.4, принадлежат ко второму поколению, их скорость отрисовки высока[2], а функциональность создана благодаря опыту работы со старыми библиотеками шаблонов.

Таблица 7.4. Движки шаблонов

Библиотека Python	Лицензия	Причины использовать
Jinja2	Лицензия BSD	• По умолчанию используется в Flask и поставляется с Django. • Основана на языке шаблонов Django Template Language, в шаблоны можно добавить лишь немного логики. • Jinja2 — это движок по умолчанию для Sphinx, Ansible и Salt (если вы использовали эти инструменты, вы знаете Jinja2)

[1] Разделение обязанностей (https://en.wikipedia.org/wiki/Separation_of_concerns) — это принцип проектирования, который означает, что хороший код является модульным: каждый компонент должен делать что-то одно.

[2] Отрисовка редко является узким местом в веб-приложениях (обычно это доступ к данным).

Библиотека Python	Лицензия	Причины использовать
Chameleon	Модифицированная лицензия BSD	• Шаблоны сами по себе являются корректными XML/HTML. • Похожа на Template Attribute Language (TAL) и его derivatives
Mako	Лицензия MIT	• Используется по умолчанию в Pyramid. • Разработана для повышения скорости выполнения программы — используйте, когда отрисовка шаблона ограничена во времени. • Позволяет поместить большое количество кода в шаблоны — Mako похож на версию Python для PHP (http://php.net/)

В следующих разделах приводится более подробная информация о библиотеках из табл. 7.4.

Jinja2

Мы рекомендуем выбирать Jinja2 (http://jinja.pocoo.org/) в качестве библиотеки шаблонов для новых веб-приложений Python. Используется в качестве движка по умолчанию в Flask и генераторе документации для Python Sphinx (http://www.sphinx-doc.org/), может применяться в Django, Pyramid и Tornado. Она работает с основанным на тексте языком шаблонов, поэтому подойдет для генерации любой разметки, а не только HTML.

Позволяет настраивать фильтры, теги, тесты и глобальные переменные. Предоставляет возможность добавлять логику в шаблоны, что сокращает объемы кода.

Рассмотрим важные теги Jinja2:

```
{# Это комментарий, он выделяется решеткой и фигурными скобками. #}
{# Так можно добавить переменную: #}
{{title}}
{# Так можно определить именованный блок, который можно заменить #}
{# на шаблон-потомок. #}
{% block head %}
<h1>This is the default heading.</h1>
{% endblock %}
{# Так можно выполнить итерирование: #}
{% for item in list %}
<li>{{ item }}</li>
{% endfor %}
```

Рассмотрим пример сайта в комбинации с веб-сервером Tornado, описанным в подразделе «Tornado» текущего раздела:

```
# импортируем Jinja2
from jinja2 import Environment, FileSystemLoader
# импортируем Tornado
import tornado.ioloop
import tornado.web
# Загружаем файл шаблона templates/site.html
TEMPLATE_FILE = "site.html"
templateLoader = FileSystemLoader( searchpath="templates/" )
templateEnv = Environment( loader=templateLoader )
template = templateEnv.get_template(TEMPLATE_FILE)
# Список популярных фильмов
movie_list = [
    [1,"The Hitchhiker's Guide to the Galaxy"],
    [2,"Back to the Future"],
    [3,"The Matrix"]
]
# Метод template.render() возвращает строку, содержащую отрисованный HTML
html_output = template.render(list=movie_list, title="My favorite movies")
# Обработчик для основной страницы
class MainHandler(tornado.web.RequestHandler):
    def get(self):
        # Возвращает отрисованную строку шаблона запросу браузера
        self.write(html_output)
# Присваиваем обработчик на сервер (127.0.0.1:PORT/)
application = tornado.web.Application([
    (r"/", MainHandler),
])
PORT=8884
if __name__ == "__main__":
    # Настраиваем сервер
    application.listen(PORT)
    tornado.ioloop.IOLoop.instance().start()
```

Файл `base.html` может быть использован в качестве основы для всех страниц сайта. В этом примере они могли бы быть реализованы в блоке `content` (в данный момент пуст):

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN">
<html lang="en">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <link rel="stylesheet" href="style.css" />
    <title>{{title}} - My Web Page</title>
</head>
<body>
<div id="content">
    {# В следующей строке будет добавлено содержимое шаблона site.html #}
    {% block content %}{% endblock %}
</div>
<div id="footer">
    {% block footer %}
    &copy; Copyright 2013 by <a href="http://domain.invalid/">you</a>.
```

```
    {% endblock %}
</div>
</body>
```

В следующем примере кода показана страница сайта (`site.html`), которая расширяет шаблон `base.html`. Блок `content` будет встроен автоматически в соответствующий блок `base.html`:

```
<!{% extends "base.html" %}
{% block content %}
    <p class="important">
    <div id="content">
        <h2>{{title}}</h2>
        <p>{{ list_title }}</p>
        <ul>
             {% for item in list %}
             <li>{{ item[0]}} :  {{ item[1]}}</li>
             {% endfor %}
        </ul>
    </div>
    </p>
{% endblock %}
```

Chameleon

Chameleon Page Templates (файлы с расширением `*.pt`) (https://chameleon.readthedocs.org/) — это реализация в движке шаблонов синтаксисов Template Attribute Language (TAL) (http://en.wikipedia.org/wiki/Template_Attribute_Language), TAL Expression Syntax (TALES) (http://bit.ly/expression-tales) и Macro Expansion TAL (Metal) (http://bit.ly/macros-metal) для HTML/XML. Chameleon анализирует Page Templates и «компилирует» их в байт-код Python для повышения скорости загрузки.

Доступен в Python 2.5 и выше (включая 3.x и PyPy) и является одним из двух движков для отрисовки, используемых Pyramid по умолчанию (см. подробнее подраздел «Pyramid» ранее в этом разделе). (Вторым является Mako, описанный в следующем подразделе.)

Page Templates добавляет специальный элемент `attributes` и текстовую разметку для вашего XML-документа: набор простых языковых конструкций позволяет управлять потоком документов, повторением элементов, заменой текста и переводом. Благодаря синтаксису, основанному на атрибутах, неотрисованные шаблоны страниц представляют собой корректный HTML, могут быть просмотрены в браузере и даже отредактированы с помощью редакторов WYSIWYG (What you see is what you get — «Что видишь, то и получаешь»). Это позволяет упростить взаимодействие с дизайнерами, а также прототипирование с использованием статических файлов. Основы языка TAL легко освоить с помощью примера:

```
<html>
  <body>
  <h1>Hello, <span tal:replace="context.name">World</span>!</h1>
    <table>
      <tr tal:repeat="row 'apple', 'banana', 'pineapple'">
        <td tal:repeat="col 'juice', 'muffin', 'pie'">
           <span tal:replace="row.capitalize()" /> <span tal:replace="col" />
        </td>
      </tr>
    </table>
  </body>
</html>
```

Шаблон `<span tal:replace="expression" />` для вставки текста достаточно распространен. Если вам не нужна стопроцентная корректность для неотрисованных шаблонов, вы можете заменить его на более сжатый и читаемый синтаксис с помощью шаблона `${expression}`, как показано далее:

```
<html>
  <body>
    <h1>Hello, ${world}!</h1>
    <table>
      <tr tal:repeat="row 'apple', 'banana', 'pineapple'">
        <td tal:repeat="col 'juice', 'muffin', 'pie'">
           ${row.capitalize()} ${col}
        </td>
      </tr>
    </table>
  </body>
</html>
```

Но помните, что полный синтаксис `<span tal:replace="expression">Default Text</span>` также позволяет использовать содержимое по умолчанию в неотрисованном шаблоне.

Chameleon не очень популярен в местах, где уже пользуются Pyramid.

Mako

Mako (http://www.makotemplates.org/) — это язык шаблонов, который компилируется в Python для максимальной производительности. Его синтаксис и API заимствованы из других языков шаблонов вроде Django и Jinja2. Этот язык шаблонов по умолчанию включается в веб-фреймворке Pyramid (рассматривается в подразделе «Pyramid» текущего раздела). Пример шаблона Mako выглядит так:

```
<%inherit file="base.html"/>
<%
    rows = [[v for v in range(0,10)] for row in range(0,10)]
%>
<table>
```

```
    % for row in rows:
        ${makerow(row)}
    % endfor
</table>
<%def name="makerow(row)">
    <tr>
    % for name in row:
        <td>${name}</td>\
    % endfor
    </tr>
</%def>
```

Это текстовый язык разметки, как Jinja2, поэтому его можно использовать для чего угодно (не только для документов XML/HTML). Чтобы отрисовать простой шаблон, введите следующий код:

```
from mako.template import Template
print(Template("hello ${data}!").render(data="world"))
```

Mako имеет хорошую репутацию в сообществе Python. Он быстр и позволяет разработчикам встраивать большое количество логики Python в страницу. Правда, мы рекомендовали бы делать это с осторожностью (только при большой необходимости).

Развертывание веб-приложений

Два варианта, которые мы рассмотрим в этом разделе, связаны либо с хостингом (то есть вы платите поставщику вроде Heroku, Gondor или PythonAnywhere, чтобы он управлял сервером и базой данных за вас), либо с настройкой собственной инфраструктуры на машине, предоставленной хостингом для виртуальных выделенных серверов (virtual private server, VPS) вроде Amazon Web Services (http://aws.amazon.com/) или Rackspace (https://www.rackspace.com/). Кратко рассмотрим оба варианта.

Хостинг

Платформа как услуга (Platform as a Service, PaaS) — это разновидность облачной вычислительной инфраструктуры, которая абстрагирует инфраструктуру и управляет ею (например, настраивает базу данных и веб-сервер, поддерживает их с помощью обновлений, связанных с безопасностью), а также отвечает за маршрутизацию и масштабирование приложений. При использовании PaaS разработчики приложений могут сфокусироваться на написании кода, а не на управлении развертыванием.

Существует множество поставщиков PaaS, конкурирующих друг с другом, но поставщики, представленные в этом списке, нацелены на сообщество Python. Большинство предлагают бесплатный тестовый период.

- *Heroku* (http://www.heroku.com/python). Рекомендуем использовать Heroku при развертывании веб-приложений Python. Он поддерживает приложения всех типов, написанные на Python 2.7–3.5: веб-приложения, серверы и фреймворки. Вам также предоставляется набор инструментов для командной строки (https://toolbelt.heroku.com/), чтобы работать с учетной записью Heroku, базой данных и веб-серверами, которые поддерживает ваше приложение, поэтому вы можете вносить изменения без веб-интерфейса. Heroku предлагает подробные статьи (https://devcenter.heroku.com/categories/python), посвященные использованию Python и Heroku, а также пошаговые инструкции (https://devcenter.heroku.com/articles/getting-started-with-python) о том, как настроить ваше первое приложение.
- *Gondor* (https://gondor.io/). Поддерживается небольшой компанией, концентрируется на том, чтобы помогать бизнесам добиваться успеха с помощью Python и Django. Его платформа предназначена для развертывания приложений Django и Pinax[1]. Платформой Gondor являются Ubuntu 12.04 и версии Django 1.4, 1.6, и 1.7, а также подмножество реализаций Python 2 и 3, перечисленных по адресу https://gondor.io/support/runtime-environment/. Он может автоматически сконфигурировать ваш сайт, написанный с помощью Django, если вы используете local_settings.py для размещения характерной для сайта информации о конфигурации. Для получения более подробных сведений обратитесь к руководству по Gondor (https://gondor.io/support/django/setup/); также доступен инструмент для командной строки (https://gondor.io/support/client/).
- *PythonAnywhere* (https://www.pythonanywhere.com/). Поддерживает Django, Flask, Tornado, Pyramid и множество других фреймворков для создания веб-приложений, которые мы не описывали, вроде Bottle (не фреймворк, как и Flask, но сообщество гораздо меньше) и web2py (отлично подходит для обучения).

  Ценовая политика хостинга связана со временем, затрачиваемым на вычисления (вместо того чтобы брать большие суммы, хостинг прерывает вычисления, как только их объем превышает дневной максимум. Это удобно для разработчиков, следящих за стоимостью).

Веб-серверы

За исключением Tornado (поставляется с собственным сервером HTTP) все фреймворки для создания веб-приложений, которые мы рассмотрели, являются приложениями WSGI. Это значит, что они должны взаимодействовать с сервером WSGI так, как это указано в PEP 3333, для того чтобы получить запрос HTTP и отправить ответ HTTP.

Большая часть самостоятельно размещаемых приложений Python сегодня находится на сервере WSGI вроде Gunicorn (сервера WSGI зачастую можно использовать как

[1] Pinax объединяет популярные шаблоны, приложения и инфраструктуру Django, чтобы ускорить запуск проекта Django.

отдельные сервера HTTP) или на базе легковесного веб-сервера вроде Nginx. Если задействуются оба варианта, серверы WSGI взаимодействуют с приложениями Python в то время, как веб-сервер обрабатывает более подходящие задачи — выдачу статических файлов, маршрутизацию запросов, защиту от распределенных атак типа отказа в обслуживании (distributed denial-of-service, DDoS) и базовую аутентификацию. Далее описываются два наиболее популярных веб-сервера — Nginx и Apache.

- *Nginx* (http://nginx.org/). Nginx (произносится «энжин-икс») — это веб-сервер и обратный прокси[1] для HTTP, SMTP и других протоколов. Отличается высокой производительностью, относительной простотой и совместимостью со многими серверами приложений (вроде серверов WSGI). Включает такие полезные функции, как балансировка нагрузки[2], базовая аутентификация, поточный режим и т. д. Nginx становится довольно популярным, поскольку разработан для обслуживания загруженных сайтов.
- *Сервер HTTP Apache*. Apache — это наиболее популярный сервер HTTP в мире (http://w3techs.com/technologies/overview/web_server/all), но мы предпочитаем Nginx. Однако те разработчики, кто только сейчас столкнулся с проблемой развертывания, могут захотеть начать с Apache и mod_wsgi (https://pypi.python.org/pypi/mod_wsgi), который считается самым простым интерфейсом WSGI. В документации к каждому фреймворку — mod_wsgi с Pyramid (http://bit.ly/pyramidwsgi), mod_wsgi с Django (http://bit.ly/django-mod_wsgi) и mod_wsgi и Flask (http://bit.ly/flask-mod_wsgi) — вы можете найти полезные руководства.

Серверы WSGI

Отдельные серверы WSGI обычно используют меньше ресурсов, чем традиционные веб-серверы, и предоставляют лучшую производительность для серверов WSGI в Python (http://nichol.as/benchmark-of-python-web-servers). Их можно использовать вместе с Nginx или Apache, которые могут служить обратными прокси. Рассмотрим наиболее популярные серверы WSGI.

- *Gunicorn (Green Unicorn)* (http://gunicorn.org/). Мы рекомендуем использовать Gunicorn для новых веб-приложений — он написан на чистом Python и применяется для обслуживания приложений Python. В отличие от других веб-серверов Python, он имеет продуманный пользовательский интерфейс, и его весьма легко сконфигурировать и использовать. Gunicorn имеет адекватные и разумные стандартные значения для конфигурации. Однако некоторые другие серверы вроде uWSGI можно настроить гораздо тщательнее (но именно поэтому работать с ними сложнее).

---

[1] Обратный прокси получает информацию от другого сервера от лица клиента и возвращает ее клиенту, как если бы она приходила от обратного прокси.

[2] Балансировка нагрузки позволяет оптимизировать производительность путем распределения работы между несколькими вычислительными ресурсами.

- *Waitress* (http://waitress.readthedocs.org/). Это написанный на чистом Python сервер WSGI. Говорят, что он имеет «очень приемлемую производительность». Его документация не особо подробна, но он предлагает такую функциональность, которой не имеет Gunicorn (например, буферизацию запросов HTTP); он не блокирует запрос, когда клиент отвечает слишком медленно (отсюда и произошло его имя Wait-ress[1]). Waitress набирает популярность среди веб-разработчиков Python.
- *uWSGI* (https://uwsgi-docs.readthedocs.org/). Предоставляет всю функциональность, необходимую для сборки хостингового сервиса. Мы не рекомендуем использовать его как отдельный веб-маршрутизатор (если только вы не уверены в том, что он вам нужен).

  uWSGI также может работать на базе полноценного веб-сервера (вроде Nginx или Apache). Веб-сервер может сконфигурировать uWSGI и работу приложения с помощью протокола uwsgi (http://bit.ly/uwsgi-protocol). Поддержка веб-сервера uWSGI позволяет динамически конфигурировать Python, передавать переменные среды и выполнять другие настройки. Для получения более подробной информации обратитесь к волшебным переменным uWSGI (http://bit.ly/uwsgimagicvar).

[1] Wait переводится как «ожидать». — *Примеч. пер.*

8 Управление кодом и его улучшение

В этой главе рассматриваются библиотеки, которые позволяют управлять процессами разработки, интеграцией систем, сервером и оптимизацией производительности, а также упрощать код.

Непрерывная интеграция

Никто не опишет процесс непрерывной интеграции лучше, чем Мартин Фаулер (Martin Fowler)[1].

> Непрерывная интеграция — это практика разработки ПО, согласно которой члены команды часто объединяют свои наработки. Обычно каждый человек делает это как минимум ежедневно, что приводит к выполнению множества интеграций за день. Каждая интеграция проверяется путем автоматической сборки (включая тесты), чтобы максимально быстро обнаружить ошибки интеграции. Многие команды считают, что такой подход значительно снижает количество проблем, возникающих при интеграции, а также позволяет быстрее разрабатывать целостность ПО.

[1] Фаулер выступает за использование правил хорошего тона при проектировании ПО. Он один из главных сторонников непрерывной интеграции (http://martinfowler.com/articles/continuousIntegration.html). Эта цитата взята из его статьи о непрерывной интеграции. Он провел серию семинаров, посвященных разработке через тестирование и отношению к экстремальной разработке (http://martinfowler.com/articles/is-tdd-dead/), вместе с Дэвидом Хайнемайером Хэнссоном (David Heinemeier Hansson, создатель Ruby on Rails) и Кентом Бэком (Kent Beck, инициатор движения экстремального программирования (XP), одним из основных принципов которого является непрерывная разработка) (https://en.wikipedia.org/wiki/Extreme_programming).

Тремя наиболее популярными инструментами для непрерывной интеграции в данный момент являются Travis-CI, Jenkins и Buildbot (перечислены в следующих разделах). Они часто используются с Tox, инструментом Python для управления virtualenv и тестами из командной строки. Travis помогает работать с несколькими интерпретаторами Python на одной платформе, а Jenkins (самый популярный) и Buildbot (написан на Python) могут управлять сборками на нескольких машинах. Многие также используют Buildout (рассмотрен в подразделе «Buildout» раздела «Инструменты изоляции» главы 3) и Docker (рассмотрен в подразделе «Docker» там же) для быстрой и частой сборки сложных сред для батареи тестов.

Tox. Tox (http://tox.readthedocs.org/en/latest/) — это инструмент автоматизации, предоставляющий функциональность упаковки, тестирования и развертывания ПО, написанного на Python, из консоли или сервера непрерывной интеграции. Является общим инструментом командной строки для управления и тестирования, который предоставляет следующие функции:

- проверка того, что все пакеты корректно устанавливаются для разных версий и интерпретаторов Python;
- запуск тестов в каждой среде, конфигурирование избранных инструментов тестирования;
- выступает в роли фронтенда для серверов непрерывной интеграции, снижая шаблонность и объединяя тесты для непрерывной интеграции и тесты для оболочки.

Вы можете установить Tox с помощью `pip`:

```
$ pip install tox
```

Системное администрирование

Инструменты, показанные в этом разделе, предназначены для наблюдения за системами и управления ими (автоматизация сервера, наблюдение за системами и управление потоком выполнения).

Travis-CI

Travis-CI (https://travis-ci.org/) — распределенный сервер непрерывной интеграции, позволяющий создавать тесты для проектов с открытым исходным кодом бесплатно. Предоставляет несколько рабочих процессов, которые запускают тесты Python, и бесшовно интегрируется с GitHub. Вы даже можете указать ему оставлять комментарии для ваших запросов на включение[1], если этот конкретный набор из-

[1] На GitHub другие пользователи оправляют запросы на включения, чтобы оповестить владельцев другого репозитория о том, что у них имеются изменения, которые они хотели бы внести в их проект.

менений сломает сборку. Поэтому, если вы размещаете свой код на GitHub, Travis-CI — отличное средство, чтобы начать использовать непрерывную интеграцию. Travis-CI может собрать ваш код на виртуальной машине, на которой запущены Linux, OS X или iOS.

Для того чтобы начать работу, добавьте файл в расширением `.travis.yml` в ваш репозиторий. В качестве примера его содержимого приведем следующий код:

```
language: python
python:
  - "2.6"
  - "2.7"
  - "3.3"
  - "3.4"
script: python tests/test_all_of_the_units.py
branches:
  only:
    - master
```

Этот код указывает протестировать ваш проект для всех перечисленных версий Python путем запуска заданного сценария, сборка будет выполнена только для ветки master. Существует множество доступных параметров вроде уведомлений, предыдущих и последующих шагов и многих других (http://about.travis-ci.org/docs/). Для того чтобы использовать Tox вместе с Travis-CI, добавьте сценарий Tox в ваш репозиторий и измените строку с конструкцией `script:`; файл должен выглядеть так:

```
install:
  - pip install tox
script:
  - tox
```

Чтобы активизировать тестирование для вашего проекта, перейдите на сайт https://travis-ci.org/ и авторизуйтесь с помощью вашей учетной записи для GitHub. Далее активизируйте ваш проект в настройках профиля, и вы готовы к работе. С этого момента тесты для вашего проекта будут запускаться после каждой отправки кода в GitHub.

Jenkins

Jenkins CI (http://jenkins.io/) — это расширяемый движок непрерывной интеграции, в данный момент он считается самым популярным. Работает на Windows, Linux и OS X, его можно подключить к «каждому существующему инструменту управления исходным кодом». Jenkins является сервлетом Java (эквивалент приложений WSGI в Java), который поставляется с собственным контейнером сервлетов, что позволяет вам запускать его с помощью команды `java --jar jenkins.war`. Для получения более подробной информации обратитесь к инструкциям по установке Jenkins (https://wiki.jenkins-ci.org/display/JENKINS/Installing+Jenkins); на странице Ubuntu содержится информация о том, как разместить Jenkins на базе обратного прокси Apache или Nginx.

Вы взаимодействуете с Jenkins с помощью информационной панели или его RESTful API[1], основанного на HTTP (например, по адресу http://myServer:8080/api), что означает, что можно использовать HTTP для того, чтобы общаться с Jenkins с удаленных машин. Например, взгляните на Jenkins Dashboard (https://builds.apache.org/) для Apache или Pylons (http://jenkins.pylonsproject.org/).

Для взаимодействия с Jenkins API в Python наиболее часто применяется python-jenkins (https://pypi.python.org/pypi/python-jenkins), созданный командой OpenStack[2] (https://www.openstack.org/). Большинство пользователей Python конфигурируют Jenkins так, чтобы он запускал сценарий Tox как часть процесса сборки. Для получения более подробной информации обратитесь к документации по адресу http://tox.readthedocs.io/en/latest/example/jenkins.html, где рассказывается об использовании Tox для Jenkins, а также к руководству http://tinyurl.com/jenkins-setup-master-slave по настройке Jenkins для работы с несколькими машинами.

Buildbot

Buildbot (http://docs.buildbot.net/current/) — это система Python, предназначенная для автоматизации цикла компиляции/тестирования, проверяющего изменения кода. Похож на Jenkins тем, что опрашивает менеджер системы контроля версий на наличие изменений, выполняет сборку и тестирование вашего кода на нескольких компьютерах в соответствии с вашими инструкциями (имеет встроенную поддержку для Tox), а затем говорит вам, что произошло. Он работает на базе веб-сервера Twisted. Если вам нужен пример того, как будет выглядеть веб-интерфейс, взгляните на общедоступную информационную панель buildbot от Chromium (https://build.chromium.org/p/chromium/waterfall) (с помощью Chromium работает браузер Chrome).

Поскольку Buildbot написан на чистом Python, его можно установить с помощью pip:

```
$ pip install buildbot
```

Версия 0.9 имеет REST API (http://docs.buildbot.net/latest/developer/apis.html), но она все еще находится на стадии бета-тестирования, поэтому вы не сможете ее использовать, если только явно не укажете номер версии (например, `pip install buildbot==0.9.00.9.0rc1`). Buildbot имеет репутацию самого мощного и самого сложного инструмента непрерывной интеграции. Для начала работы с ним обратитесь к этому отличному руководству: http://docs.buildbot.net/current/tutorial.

---

[1] REST расшифровывается как representational state transfer — передача состояния представления. Это не стандарт и не протокол, а набор принципов проектирования, разработанных во время создания стандарта HTTP 1.1. Список релевантных архитектурных ограничений для REST доступен в «Википедии»: https://en.wikipedia.org/wiki/Representational_state_transfer#Architectural_constraints.

[2] OpenStack предоставляет бесплатное ПО для облачных сетей, хранения и вычислений, поэтому организации могут размещать собственные приватные облака или публичные облака, за доступ к которым нужно платить.

Автоматизация сервера

Salt, Ansible, Puppet, Chef и CFEngine — это инструменты для автоматизации сервера, которые предоставляют системным администраторам элегантный способ управлять их флотом физических и виртуальных машин. Все они могут управлять машинами, на которых установлены Linux, Unix-подобные системы, а также Windows.

Конечно, мы можем использовать только Salt и Ansible, поскольку они написаны на Python. Но они относительно новые, другие же варианты применяются более широко. В следующих разделах проведен их краткий обзор.

Разработчики Docker ожидают, что инструменты по автоматизации систем вроде Salt, Ansible и прочих будут дополнены Docker, а не заменены им. Взгляните на статью http://stackshare.io/posts/how-docker-fits-into-the-current-devops-landscape о том, как Docker работает с остальным ПО для DevOps.

Salt

Salt (http://saltstack.org/) называет свой главный узел *мастером*, а узлы-агенты — *миньонами* или *хостами-миньонами*. Его основная цель — высокая скорость; работа с сетью по умолчанию выполняется с помощью ZeroMQ, между мастером и миньонами устанавливается соединение TCP. Члены команды разработчиков Salt написали свой (необязательный) протокол передачи данных RAET (https://github.com/saltstack/raet), который работает быстрее, чем TCP, и теряет не так много данных, как UDP.

Salt поддерживает версии Python 2.6 и 2.7, его можно установить с помощью команды `pip`:

```
$ pip install salt  # Для Python 3 версии пока нет...
```

После конфигурирования сервера-мастера и любого количества хостов-миньонов мы можем запускать для миньонов произвольные команды оболочки или использовать заранее созданные модули, состоящие из сложных команд. Следующая команда перечисляет всех доступных миньонов с помощью команды `ping` из модуля тестирования salt:

```
$ salt '*' test.ping
```

Вы можете отфильтровать хосты-миньоны либо по их идентификатору, либо с помощью системы grains (http://docs.saltstack.org/en/latest/topics/targeting/grains.html), которая использует статическую информацию хоста вроде версии операционной системы или архитектуры ЦП, чтобы предоставить таксономию хостов для модулей Salt. Например, следующая команда применяет систему grains для перечисления только тех миньонов, на которых запущена CentOS:

```
$ salt -G 'os:CentOS' test.ping
```

Salt также предоставляет систему состояний. Состояния могут быть использованы для конфигурирования хостов-миньонов. Например, когда миньону указывается прочесть следующий файл состояний, он установит и запустит сервер Apache:

```
apache:
  pkg:
    - installed
  service:
    - running
    - enable: True
    - require:
      - pkg: apache
```

Файлы состояний могут быть написаны с помощью YAML, дополненной системой шаблонов Jinja2, или же могут быть чистыми модулями Python. Для получения более подробной информации обратитесь к документации Salt по адресу http://docs.saltstack.com/.

Ansible

Самое большое преимущество Ansible (http://ansible.com/) перед другими системами автоматизации — для ее установки на клиентских машинах не требуется ничего (кроме Python). Все другие варианты[1] поддерживают на клиентах демонов, которые опрашивают мастера. Их конфигурационные файлы имеют формат YAML. *Сценарии* — это документы для конфигурирования, развертывания и управления для Ansible, они написаны на YAML и используют для шаблонов язык Jinja2. Ansible поддерживает версии Python 2.6 и 2.7, ее можно установить с помощью `pip`:

```
$ pip install ansible  # Для Python 3 версии пока нет...
```

Ansible требует наличия файла инвентаря, в котором описывается, к каким хостам он имеет доступ. В следующем фрагменте кода показывается пример хоста и сценария, который опрашивает все хосты в файле инвентаря. Рассмотрим пример файла инвентаря (`hosts.yml`):

```
[server_name]
127.0.0.1
Рассмотрим пример сценария (ping.yml):
---
- hosts: all
  tasks:
    - name: ping
      action: ping
```

[1] За исключением Salt-SSH, который является альтернативной архитектурой Salt (возможно, она была создана как ответ пользователям, желавшим получить версию Salt, более похожую на Ansible).

Для того чтобы запустить сценарий, введите следующую команду:

```
$ ansible-playbook ping.yml -i hosts.yml --ask-pass
```

Сценарий Ansible будет опрашивать все серверы, перечисленные в файле `hosts.yml`. С помощью Ansible вы можете выбрать группы серверов. Для получения более подробной информации об Ansible прочтите документацию к ней (http://docs.ansible.com/). Руководство по Ansible Servers for Hackers (https://serversforhackers.com/an-ansible-tutorial/) также содержит подробную информацию.

Puppet

Puppet написан на Ruby и предоставляет собственный язык — PuppetScript — для конфигурирования. Имеет выделенный сервер Puppet Master, который отвечает за управление *узлами-агентами*. *Модули* — это небольшие разделяемые единицы кода, написанные для автоматизации или определения состояния системы. Puppet Forge (https://forge.puppetlabs.com/) — это репозиторий для модулей, написанных сообществом для Open Source Puppet и Puppet Enterprise.

Узлы-агенты отправляют основную информацию о системе (например, операционную систему, ядро, архитектуру, IP-адрес и имя хоста) Puppet Master. Puppet Master компилирует на ее основе каталог данных о том, как нужно конфигурировать каждый узел, и пересылает его агенту. Агент выполняет изменения на основе того, что указано в каталоге, и отправляет Puppet Master отчет о проделанной работе.

Facter (да, его имя заканчивается на -er) — весьма интересный инструмент, который поставляется с Puppet и получает основную информацию о системе. Вы можете сослаться на эти факты как на переменные при написании модулей Puppet:

```
$ facter kernel
Linux
$
$ facter operatingsystem
Ubuntu
```

Процесс написания модулей в Puppet довольно прямолинеен: манифесты Puppet (файлы с расширением `*.pp`) формируют модули Puppet. Рассмотрим пример приложения Hello World для Puppet:

```
notify { 'Hello World, this message is getting logged into the agent node':
    #As nothing is specified in the body, the resource title
    #is the notification message by default.
}
```

Перед вами еще один пример, использующий логику системы. Чтобы обратиться к другим фактам, добавьте знак `$` к имени переменной, например `$hostname` (в нашем случае `$operatingsystem`):

```
notify{ 'Mac Warning':
    message => $operatingsystem ? {
        'Darwin' => 'This seems to be a Mac.',
        default  => 'I am a PC.',
    },
}
```

Для Puppet существует несколько типов ресурсов, но парадигма «пакет-файл-сервис» — это все, что вам нужно для выполнения большинства задач по управлению конфигурацией. Следующий код Puppet позволяет убедиться, что пакет OpenSSH-Server устанавливается в системе. Сервису sshd (демон сервера SSH) указывается выполнять перезапуск каждый раз, когда изменяется конфигурационный файл sshd:

```
package { 'openssh-server':
    ensure => installed,
}
file { '/etc/ssh/sshd_config':
    source   => 'puppet:///modules/sshd/sshd_config',
    owner    => 'root',
    group    => 'root',
    mode     => '640',
    notify   =>  Service['sshd'], # sshd перезапустится
                                  # после каждого изменения этого файла
    require  => Package['openssh-server'],
}

service { 'sshd':
    ensure    => running,
    enable    => true,
    hasstatus => true,
    hasrestart=> true,
}
```

Для получения более подробной информации обратитесь к документации к Puppet Labs (http://docs.puppetlabs.com/).

Chef

Если для управления конфигурацией вы выбираете Chef (https://www.chef.io/chef/), то для написания кода инфраструктуры будете использовать язык Ruby. Chef похож на Puppet, но разработан с противоположной философией: Puppet предоставляет фреймворк, который упрощает работу за счет гибкости, а Chef практически не предоставляет фреймворка (его цель — быть очень гибким, поэтому его сложнее использовать).

Клиенты Chef работают на каждом узле вашей инфраструктуры и регулярно опрашивают *сервер* Chef, чтобы гарантировать, что ваша система всегда находится в рабочем состоянии.

Каждый отдельный клиент Chef конфигурирует себя самостоятельно. Такой подход делает Chef масштабируемой платформой по автоматизации.

Chef для работы использует пользовательские *рецепты* (элементы конфигурации), реализованные в `cookbooks.Cookbooks`, которые, по сути, являются пакетами для инфраструктурного выбора и обычно хранятся на сервере Chef. Прочтите серию руководств от DigitalOcean, посвященных Chef (http://tinyurl.com/digitalocean-chef-tutorial), чтобы узнать, как создать просто сервер Chef.

Используйте команду `knife` для создания поваренной книги:

```
$ knife cookbook create cookbook_name
```

Статья Энди Гейла (Andy Gale) Getting started with Chef (http://gettingstartedwith-chef.com/first-steps-with-chef.html) — хорошая стартовая точка для тех, кто начинает работать с Chef.

Множество поваренных книг сообщества вы можете найти в Chef Supermarket — с их помощью вы легко сможете начать писать собственные поваренные книги. Для получения более подробной информации обратитесь к полной документации Chef (https://docs.chef.io/).

CFEngine

CFEngine имеет крошечный отпечаток, поскольку написан на C. Основная цель ее проекта — отказоустойчивость. Она достигается с помощью автономных агентов, работающих в распределенной сети (в противоположность архитектуре мастер/клиент), которые общаются с использованием теории обещаний (https://en.wikipedia.org/wiki/Promise_theory). Если вам нужна архитектура без мастера, попробуйте эту систему.

Наблюдение за системами и задачами

Следующие библиотеки помогут всем системным администраторам наблюдать запущенные задачи, но их приложения значительно отличаются друг от друга: Psutil предоставляет информацию в Python, которая может быть получена вспомогательными функциями Unix, Fabric позволяет легко определить и выполнить команды для набора удаленных хостов с помощью SSH, а Luigi помогает планировать запуск и наблюдение за долгоиграющими пакетными процессами вроде цепочек команд Hadoop.

Psutil

Psutil (https://pythonhosted.org/psutil/) — это кросс-платформенный (включая Windows) интерфейс для разного рода системной информации (например, ЦП, памяти, дисков,

сети, пользователей и процессов). Позволяет с помощью Python получить доступ к информации, которую многие из нас привыкли получать, используя команды Unix (https://en.wikipedia.org/wiki/List_of_Unix_commands) вроде `top`, `ps`, `df` и `netstat`. Установите его с помощью `pip`:

```
$ pip install psutil
```

Рассмотрим пример, который наблюдает за перегрузкой сервера (если какой-то тест — сети или ЦП — даст сбой, он отправит электронное письмо):

```
# Функции для получения значений системы:
from psutil import cpu_percent, net_io_counters
# Функции для перерыва:
from time import sleep
# Пакет для сервисов электронной почты:
import smtplib
import string
MAX_NET_USAGE = 400000
MAX_ATTACKS = 4
attack = 0
counter = 0
while attack <= MAX_ATTACKS:
    sleep(4)
    counter = counter + 1
    # Проверяем использование ЦП
    if cpu_percent(interval = 1) > 70:
        attack = attack + 1
    # Проверяем использование сети
    neti1 = net_io_counters()[1]
    neto1 = net_io_counters()[0]
    sleep(1)
    neti2 = net_io_counters()[1]
    neto2 = net_io_counters()[0]
    # Рассчитываем байты в секунду
    net = ((neti2+neto2) - (neti1+neto1))/2
    if net > MAX_NET_USAGE:
        attack = attack + 1
    if counter > 25:
        attack = 0
        counter = 0
# Пишем очень важное электронное письмо, если значение параметра attack больше 4
TO = "you@your_email.com"
FROM = "webmaster@your_domain.com"
SUBJECT = "Your domain is out of system resources!"
text = "Go and fix your server!"
BODY = string.join(
     ("From: %s" %FROM,"To: %s" %TO,"Subject: %s" %SUBJECT, "",text), "\r\n")
server = smtplib.SMTP('127.0.0.1')
server.sendmail(FROM, [TO], BODY)
server.quit()
```

Хорошим примером использования Psutil является glances (https://github.com/nicolargo/glances/) — полностью консольное приложение, которое ведет себя как расширенная версия `top` (перечисляет запущенные процессы, упорядочивая их по использованию ЦП или другим способом, указанным пользователем) и имеет возможность наблюдать за клиентом и сервером.

Fabric

Fabric (http://docs.fabfile.org/) — это библиотека, предназначенная для упрощения задач системного администратора. Позволяет соединяться с помощью SSH с несколькими хостами и выполнять задачи для каждого из них. Это удобно для системных администраторов, а также для тех, кто развертывает приложения. Чтобы установить Fabric, используйте `pip`:

```
$ pip install fabric
```

Рассмотрим полный модуль Python, определяющий две задачи Fabric: `memory_usage` и `deploy`:

```
# fabfile.py
from fabric.api import cd, env, prefix, run, task
env.hosts = ['my_server1', 'my_server2']   # С чем соединяться по SSH
@task
def memory_usage():
    run('free -m')
@task
def deploy():
    with cd('/var/www/project-env/project'):
        with prefix('. ../bin/activate'):
            run('git pull')
            run('touch app.wsgi')
```

Оператор `with` вкладывает команды друг в друга, поэтому в конечном счете метод `deploy()` для каждого хоста начинает выглядеть так:

```
$ ssh имя_хоста cd /var/ww/project-env/project && ../bin/activate && git pull
$ ssh имя_хоста cd /var/ww/project-env/project && ../bin/activate && \
> touch app.wsgi
```

Учитывая предыдущий код, сохраненный в файле `fabfile.py` (имя модуля по умолчанию, которое ищет fab), мы можем проверить использование памяти с помощью нашей новой задачи `memory_usage`:

```
$ fab memory_usage
[my_server1] Executing task 'memory'
[my_server1] run: free -m [my_server1] out:              total     used     free
shared  buffers   cached
[my_server1] out: Mem:          6964     1897     5067        0      166      222
```

```
[my_server1] out: -/+ buffers/cache:     1509     5455
[my_server1] out: Swap:            0        0        0
[my_server2] Executing task 'memory'
[my_server2] run: free -m [my_server2] out:              total     used     free
shared  buffers   cached
[my_server2] out: Mem:          1666      902      764        0      180      572
[my_server2] out: -/+ buffers/cache:      148     1517
[my_server2] out: Swap:          895        1      894
```

И мы можем выполнить развертывание:

```
$ fab deploy
```

Дополнительная функциональность включает параллельное исполнение, взаимодействие с удаленными программами и группирование хостов. В документации к Fabric (http://docs.fabfile.org/) приведены понятные примеры.

Luigi

Luigi (https://pypi.python.org/pypi/luigi) — это инструмент для управления конвейером, разработанный и выпущенный компанией Spotify. Помогает разработчикам управлять целым конвейером крупных долгоиграющих пакетных задач, объединяя запросы Hive, запросы к базе данных, задачи Hadoop Java, задачи pySpark и многие другие задачи, которые вы можете написать самостоятельно. Они не должны быть приложениями, работающими с большими данными, API позволяет запланировать что угодно. Spotify позволил запускать задачи с помощью Hadoop, поэтому разработчики предоставляют все необходимые вспомогательные программы в пакете `luigi.contrib` (http://luigi.readthedocs.io/en/stable/api/luigi.contrib.html). Установите его с помощью `pip`:

```
$ pip install luigi
```

Включает в себя веб-интерфейс, поэтому пользователи могут фильтровать свои задачи и просматривать графики зависимостей для рабочего потока конвейера и их продвижение. В репозитории GitHub вы можете найти примеры задач Luigi, также можете просмотреть документацию к Luigi.

Скорость

В этом разделе перечислены наиболее популярные способы оптимизации скорости, используемые сообществом Python. В табл. 8.1 показаны варианты оптимизации, которые вы можете применить после того, как попробуете простые методы вроде запуска профайлера (https://docs.python.org/3.5/library/profile.html) и сравнения параметров для сниппетов кода (https://docs.python.org/3.5/library/timeit.html).

Вы, возможно, уже слышали о глобальной блокировке интерпретатора (global interpreter lock, GIL) (http://wiki.python.org/moin/GlobalInterpreterLock). Это способ, с по-

мощью которого реализация C для Python позволяет нескольким потокам работать одновременно.

Управление памятью в Python не полностью потокобезопасно, поэтому GIL нужен для того, чтобы помешать нескольким потокам запускать одновременно один и тот же код.

GIL зачастую называют ограничением Python, но он не является такой уж большой проблемой — мешает лишь тогда, когда процессы связаны с ЦП (в этом случае, как и для NumPy или криптографических библиотек, которые мы рассмотрим далее, код переписан на C со связыванием с Python). Для всего прочего (для ввода/вывода в сетях или файлах) узким местом является код, блокирующий один поток, ожидающий завершения операции ввода/вывода. Вы можете решить проблему с блокировкой с помощью потоков или событийно-ориентированного программирования.

Отметим, что в Python 2 имеются медленная и быстрая версии библиотек — StringIO и cStringIO, ElementTree и cElementTree. Реализации на C работают быстрее, но их нужно явно импортировать. Начиная с версии Python 3.3 обычные версии импортируют быструю реализацию там, где это возможно, а библиотеки, чье имя начинается с C, считаются устаревшими.

Таблица 8.1. Способы ускорения работы

Способ	Лицензия	Причины использовать
Многопоточность	PSFL	• Позволяет создавать несколько потоков выполнения. • Многопоточность (при использовании CPython из-за наличия GIL) не задействует собственные процессы; разные потоки переключаются, когда один из них заблокирован (это полезно, когда узким местом является какая-нибудь блокирующая задача вроде ожидания окончания операции ввода/вывода). • GIL отсутствует в некоторых других реализациях Python вроде Jython и IronPython
Multiprocessing/subprocess	PSFL	• Инструменты библиотеки multiprocessing позволяют создавать другие процессы Python, минуя GIL. • Subprocess позволяет запускать несколько процессов командной строки
PyPy	Лицензия MIT	• Этот интерпретатор Python (в данный момент для версии Python 2.7.10 или 3.2.5) предоставляет возможность динамической компиляции в код на языке C там, где это возможно.

Продолжение ⇗

Таблица 8.1 *(продолжение)*

Способ	Лицензия	Причины использовать
		• Не требует усилий: вам не нужно писать код, при этом он дает значительный прирост скорости. • Полноценная замена для CPython, которая обычно работает (любые библиотеки, написанные на C, должны использовать CFFI или находиться в списке совместимости PyPy (http://pypy.org/compat.html))
Cython	Лицензия Apache	Предоставляет два способа статически скомпилировать код Python: использование языка аннотаций Cython (*.pxd); статистическое компилирование кода на чистом Python и применение декораторов Cython для указания типа объекта
Numba	Лицензия BSD	• Предоставляет статический (благодаря инструменту pycc) и динамический компиляторы, компилирующие код в машинный код. Использует массивы NumPy. • Требуется версия Python 2.7 или 3.4, библиотека llvmlite (http://llvmlite.pydata.org/en/latest/install/index.html) и ее зависимость, инфраструктура LLVM (Low-Level Virtual Machine)
Weave	Лицензия BSD	• Предоставляет способ «сплести» несколько строк кода на C в код на Python (применяйте его только в том случае, если вы уже используете Weave). • В противном случае используйте Cython — Weave считается устаревшим
PyCUDA/gnumpy/TensorFlow/Theano/PyOpenCL	MIT/модифицированная лицензия BSD/BSD/BSD/MIT	• Эти библиотеки предоставляют разные способы использования NVIDIA GPU, если он у вас установлен, и могут установить набор инструментов для CUDA (http://docs.nvidia.com/cuda/). • PyOpenCL может использовать процессоры не только от NVIDIA. • Для каждого из них существует собственное приложение, например gnumpy предполагается как полноценная замена для NumPy
Непосредственное использование библиотек C/C++	—	Повышение скорости стоит того, чтобы потратить дополнительное время на написание кода на C/C++

Джефф Напп (Jeff Knupp), автор книги *Writing Idiomatic Python* (http://bit.ly/writing-idiomatic-python), написал статью о том, как обойти GIL (http://bit.ly/pythons-hardest-problems), процитировав статью Дэвида Бизли (David Beazley)[1] на эту тему.

Многопоточность и другие способы оптимизации, показанные в табл. 8.1, более подробно рассматриваются в следующих разделах.

Многопоточность

Библиотека для работы с потоками в Python позволяет создать несколько потоков. Из-за GIL (во всяком случае в CPython) только один процесс Python может быть запущен для каждого интерпретатора; это означает, что прирост производительности появится только в том случае, если хотя бы один поток заблокирован (например, для ввода/вывода). Еще один вариант для ввода/вывода — обработка событий. Чтобы узнать подробнее, прочтите абзацы, связанные с asyncio, в подразделе «Производительность сетевых инструментов из стандартной библиотеки Python» раздела «Распределенные системы» главы 9.

Когда у вас есть несколько потоков Python, ядро замечает, что один из потоков заблокирован для ввода-вывода, и оно переключается, чтобы позволить следующему потоку использовать процессор до тех пор, пока он не будет заблокирован или не завершится. Все это происходит автоматически, когда вы запускаете ваши потоки. Есть хороший пример применения многопоточности на сайте Stack Overflow (http://bit.ly/threading-in-python), для серии Python Module of the Week написана отличная статья на тему многопоточности (https://pymotw.com/2/threading/). Вы также можете просмотреть документацию стандартной библиотеки, посвященную работе с потоками (https://docs.python.org/3/library/threading.html).

Модуль multiprocessing

Модуль multiprocessing (https://docs.python.org/3/library/multiprocessing.html) стандартной библиотеки Python предоставляет способ обойти GIL путем запуска дополнительных интерпретаторов Python. Отдельные процессы могут общаться друг с другом с помощью запросов `multiprocessing.Pipe` или `multiprocessing.Queue`; также они могут делиться памятью с помощью запросов `multiprocessing.Array` или `multiprocessing.Value`, что автоматически реализует блокировки. Осторожно делитесь данными; эти объекты реализуют блокировку для того, чтобы предотвратить одновременный доступ разных процессов.

---

[1] Дэвид Бизли написал отличное руководство (http://www.dabeaz.com/python/UnderstandingGIL.pdf), которое описывает способ работы GIL. Он также рассматривает новую версию GIL (http://www.dabeaz.com/python/NewGIL.pdf), появившуюся в Python 3.2. Результаты показывают, что максимизация производительности в приложении Python требует глубокого понимания GIL, его влияния на приложение, а также знания количества ядер и понимания узких мест приложения.

Рассмотрим пример, иллюстрирующий прирост скорости, который появляется в результате использования пула работников. Существует компромисс между сэкономленным временем и временем, затрачиваемым на переключение между интерпретаторами. В примере используется метод Монте-Карло для оценки значения числа пи[1]:

```
>>> import multiprocessing
>>> import multiprocessing
>>> import random
>>> import timeit
>>>
>>> def calculate_pi(iterations):
...     x = (random.random() for i in range(iterations))
...     y = (random.random() for i in range(iterations))
...     r_squared = [xi**2 + yi**2 for xi, yi in zip(x, y)]
...     percent_coverage = sum([r <= 1 for r in r_squared]) /
        / len(r_squared)
...     return 4 * percent_coverage
...
>>>
>>> def run_pool(processes, total_iterations):
...     with multiprocessing.Pool(processes) as pool: (1)
...         # Разделим общее количество итераций между процессами. (2)
...         iterations = [total_iterations // processes] *
            * processes
...         result = pool.map(calculate_pi, iterations) (3)
...     print( "%0.4f" % (sum(result) / processes), end=',  ')
...
>>>
>>> ten_million = 10000000 (4)
>>> timeit.timeit(lambda: run_pool(1, ten_million), number=10)
3.141,  3.142,  3.142,  3.141,  3.141,  3.142,  3.141,  3.141, 3.142,
3.142, 134.48382110201055 (5)
>>>                          (6)
>>> timeit.timeit(lambda: run_pool(10, ten_million), number=10)
3.142,  3.142,  3.142,  3.142,  3.142,  3.142,  3.141,  3.142,
3.142,  3.141, 74.38514468498761 (7)
```

(1) Использование `multiprocessing.Pool` внутри менеджера контекста указывает, что пул должен применяться только тем процессом, который его создал.

(2) Общее количество итераций останется неизменным, оно лишь будет поделено на разное количество процессов.

[1] По адресу http://bit.ly/monte-carlo-pi приводится полная реализация метода. По сути, вы бросаете дротики в квадрат размером 2 × 2, а круг имеет радиус = 1. Если дротик может попасть в любое место доски с одинаковой вероятностью, то процент дротиков, попавших в круг, будет равен $\pi / 4$. Это означает, что если вы умножите шанс попадания в круг на 4, то получите число π.

❸ Метод `pool.map()` создает несколько процессов — по одному на каждый элемент списка итераций; максимальное количество равно числу, указанному при инициализации пула (в вызове `multiprocessing.Pool(processes)`).

❹ Существует только один процесс для первого испытания `timeit`.

❺ 10 повторений одного процесса с 10 миллионами итераций заняли 134 секунды.

❻ Для второго испытания `timeit` создано 10 процессов.

❼ 10 повторений десяти процессов, каждый из которых имеет один миллион итераций, заняли 74 секунды.

Идея заключается в том, что существуют накладные расходы при создании нескольких процессов, но инструменты, позволяющие запустить с помощью Python несколько процессов, довольно надежны. Для получения более подробной информации просмотрите документацию о библиотеке multiprocessing в стандартной библиотеке (https://docs.python.org/3.5/library/multiprocessing.html), а также прочитайте статью Джеффа Наппа (Jeff Knupp) о том, как обойти GIL (пара абзацев посвящены этой библиотеке) (http://bit.ly/pythons-hardest-problems).

Subprocess

Библиотека subprocess (https://docs.python.org/3/library/subprocess.html) была представлена в версии стандартной библиотеки для Python 2.4, определена в PEP 324 (https://www.python.org/dev/peps/pep-0324). Выполняет системный вызов (вроде `unzip` или `curl`), как если бы она была вызвана из командной строки (по умолчанию, не вызывая системную оболочку (http://bit.ly/subprocess-security)), а разработчик выбирает, что нужно сделать с входным и выходным конвейерами subprocess. Мы рекомендуем пользователям Python 2 получить обновленную версию пакета subprocess32, в которой исправляются некоторые ошибки. Установите его с помощью `pip`:

```
$ pip install subprocess32
```

В блоге Python Module of the Week вы можете найти отличное руководство по subprocess (https://pymotw.com/2/subprocess/).

PyPy

PyPy — это реализация Python на чистом Python. Она быстра; и когда она работает, вам не нужно больше ничего делать со своим кодом — он будет работать без дополнительных усилий. Вам следует воспользоваться этим вариантом в первую очередь.

Вы не можете получить ее с помощью команды `pip`, поскольку, по сути, это еще одна реализация Python. На странице загрузок PyPy http://pypy.org/download.html

вы можете найти корректную версию реализации для вашей версии Python и операционной системы.

Существует модифицированная версия тестового кода, ограниченная по процессору, от Дэвида Бизли (David Beazley) (http://www.dabeaz.com/GIL/gilvis/measure2.py), в которую добавлен цикл для выполнения нескольких тестов. Вы можете увидеть разницу между PyPy и CPython.

Сначала запустим ее с помощью CPython:

```
$ # CPython
$ ./python -V Python 2.7.1
$
$ ./python measure2.py
1.06774401665 1.45412397385 1.51485204697 1.54693889618 1.60109114647
```

А теперь запустим тот же сценарий, но изменим интерпретатор Python — выберем PyPy:

```
$ # PyPy
$ ./pypy -V Python 2.7.1 (7773f8fc4223, Nov 18 2011, 18:47:10)
 [PyPy 1.7.0 with GCC 4.4.3]
$
$ ./pypy measure2.py
0.068399906158 0.048321008682 30.038858890533 40.044069051742 60.069530010223 4
```

Получается, что благодаря простой загрузке PyPy мы сократили время работы сценария с 1,4 секунды до 0,05 — практически в 20 раз. Порой ваш код будет ускорен менее чем в два раза, но иногда вы сможете значительно его ускорить. И для этого не нужно прикладывать никаких усилий, за исключением загрузки интерпретатора PyPy. Если хотите, чтобы ваша библиотека, написанная на C, была совместима с PyPy, следуйте советам PyPy (http://pypy.org/compat.html) и используйте CFFI вместо ctypes из стандартной библиотеки.

Cython

К сожалению, PyPy работает не со всеми библиотеками, использующими расширения, написанные на C. Для этих случаев Cython (произносится «сайтон» — это не то же самое, что CPython, стандартная реализация Python, созданная с помощью C) (http://cython.org/) реализует superset языка Python (можно писать модули Python на C и C++). Cython дает возможность вызывать функции из скомпилированных библиотек на C и предоставляет контекст `nogil`, позволяющий обойти GIL для раздела кода (он не манипулирует объектами Python) (http://tinyurl.com/cython-nogil). Применяя Cython, вы можете воспользоваться преимуществами строгого типизирования в Python[1] переменных и операций.

[1] Язык может быть одновременно строго типизированным и динамически типизированным, это описывается по адресу http://stackoverflow.com/questions/11328920/.

Рассмотрим пример строгой типизации с помощью Cython:

```
def primes(int kmax):
"""Расчет простых чисел с помощью дополнительных ключевых слов Cython """
    cdef int n, k, i
    cdef int p[1000]
    result = []
    if kmax > 1000:
         kmax = 1000
    k = 0
    n = 2
    while k < kmax:
        i = 0
        while i < k and n % p[i] != 0:
            i = i + 1
        if i == k:
            p[k] = n
            k = k + 1
            result.append(n)
        n = n + 1
    return result
```

Эта реализация алгоритма поиска простых чисел содержит дополнительные ключевые слова. Следующий пример написан на чистом Python:

```
def primes(kmax):
""" Расчет простых чисел с помощью стандартного синтаксиса Python """
    p= range(1000)
    result = []
    if kmax > 1000:
        kmax = 1000
    k = 0
    n = 2
    while k < kmax:
        i = 0
        while i < k and n % p[i] != 0:
            i = i + 1
        if i == k:
            p[k] = n
            k = k + 1
            result.append(n)
        n = n + 1
    return result
```

Обратите внимание: в версии Cython вы объявляете, что целые числа и массивы целых чисел будут скомпилированы в типы C, в то же время будет создан список Python:

```
# Версия Cython

def primes(int kmax): (1)
    """ Расчет простых чисел с помощью дополнительных
    ключевых слов Cython """
    cdef int n, k, i  (2)
```

```
    cdef int p[1000]  ❸
    result = []
```

❶ Тип объявляется как целое число.

❷ Переменные `n`, `k` и `i` объявляются как целые числа.

❸ Мы заранее выделяем память для массива целых чисел p размером в 1000 элементов.

В чем же разница? В версии Cython вы можете увидеть объявление типов переменных и массива целых чисел, которые выглядят так же, как и в обычном C. Например, дополнительное объявление типа (целочисленного) в выражении `cdef int n,k,i` позволяет компилятору Cython генерировать более эффективный код C. Поскольку синтаксис несовместим со стандартным Python, он сохраняется в файлах с расширением `*.pyx`, а не с расширением `*.py`.

Каковы различия в скорости? Давайте проверим!

```
import time
# активизируем компилятор pyx
import pyximport  ❶
pyximport.install()  ❷
# primes, реализованный с помощью Cython
import primesCy
# primes, реализованный с помощью Python
import primes

print("Cython:")
t1 = time.time()
print primesCy.primes(500)
t2 = time.time()
print("Cython time: %s" %(t2-t1))
print("")
print("Python")
t1 = time.time()  ❸
print(primes.primes(500))
t2 = time.time()
print("Python time: {}".format(t2-t1))
```

❶ Модуль `pyximport` позволяет импортировать файлы с расширением `*.pyx` (например, `primesCy.pyx`) с помощью скомпилированной в Cython версии функции `primes`.

❷ Команда `pyximport.install()` позволяет интерпретатору Python непосредственно запустить компилятор Cython для генерации кода C, который автоматически компилируется в библиотеку с расширением `*.so`. Далее Cython может легко и эффективно импортировать эту библиотеку в ваш код Python.

❸ С помощью функции `time.time()` вы можете сравнить время выполнения этих двух вызовов, которые определяют 500 простых чисел. На стандартном ноутбуке (dual-core AMD E-450 1.6 GHz) мы получили следующие значения:

```
Cython time: 0.0054 seconds
Python time: 0.0566 seconds
```

А здесь результат работы встроенной машины ARM BeagleBone (http://beagleboard.org/Products/BeagleBone):

```
Cython time: 0.0196 seconds
Python time: 0.3302 seconds
```

Numba

Numba (http://numba.pydata.org/) — это компилятор для Python, поддерживающий NumPy (он является динамическим — just-in-time (JIT)). Компилирует аннотированный код Python (и NumPy) для LLVM (Low-Level Virtual Machine) (http://llvm.org/) с помощью особых декораторов. Вкратце, Numba использует LLVM для компилирования кода Python в машинный код, который может быть нативно выполнен во время работы программы.

Если вы используете Anaconda, установите Numba с помощью команды `conda install numba`. Если нет, установите его вручную. Вы должны заранее установить NumPy и LLVM (перед Numba).

Проверьте, какая версия LLVM вам нужна (на странице PyPI для llvmlite по адресу https://pypi.python.org/pypi/llvmlite), и загрузите ее для вашей ОС:

- сборки LLVM для Windows (http://llvm.org/builds/);
- сборки LLVM для Debian/Ubuntu (http://llvm.org/apt/);
- сборки LLVM для Fedora (https://apps.fedoraproject.org/packages/llvm);
- вы можете найти информацию о том, как устанавливать LLVM на основе исходного кода для других систем Unix, в разделе «Сборка компиляторов Clang + LLVM» (http://ftp.math.utah.edu/pub/llvm/);
- для OS X используйте команду `brew install homebrew/versions/llvm37` (или выберите текущую версию).

Как только вы установите LLVM и NumPy, инсталлируйте Numba с помощью `pip`. Вам может понадобиться помочь установщику найти файл `llvm-config`, предоставив переменной среды `LLVM_CONFIG` соответствующий путь, например:

```
$ LLVM_CONFIG=/path/to/llvm-config-3.7 pip install numba
```

Чтобы использовать его в своем коде, декорируйте свои функции:

```
from numba import jit, int32

@jit ❶
def f(x):
    return x + 3
```

```
@jit(int32(int32, int32))  ❷
def g(x, y):
    return x + y
```

❶ Без аргументов декоратор `@jit` выполняет ленивую компиляцию — сам решает, оптимизировать ли функцию и как это сделать.

❷ Для ранней компиляции укажите типы. Функция будет скомпилирована с указанной специализацией, ни одну другую специализацию использовать не получится, возвращаемое значение и два аргумента будут иметь тип `numba.int32`.

Флаг `nogil` позволит коду игнорировать глобальную блокировку интерпретатора, а модуль `numba.pycc` можно использовать для компилирования кода заранее. Для получения более подробной информации обратитесь к руководству пользователя для Numba (http://numba.pydata.org/numba-doc/latest/user).

Библиотеки для работы с GPU

Numba опционально может быть создан с той производительностью, которая позволит ему работать на графическом процессоре (graphics processing unit, GPU), оптимизированном для выполнения быстрых параллельных вычислений в современных видеоиграх. Вам понадобится NVIDIA GPU, также нужно установить тулкит CUDA Toolkit от NVIDIA (https://developer.nvidia.com/cuda-downloads). Далее следуйте инструкциям документации по использованию Numba's CUDA JIT для GPU (http://numba.pydata.org/numba-doc/0.13/CUDAJit.html).

Помимо Numba, еще одной популярной библиотекой, которая может работать с GPU, является TensorFlow (https://www.tensorflow.org/). Выпущена компанией Google под лицензией Apache v2.0. Предоставляет возможность использовать тензоры (многомерные матрицы), а также способ объединить в цепочку операции над ними для более быстрого выполнения операций над матрицами.

На данный момент она может использовать GPU только в операционных системах Linux. Для получения более подробных инструкций обратитесь к следующим страницам:

- установка TensorFlow с поддержкой GPU — http://bit.ly/tensorflow-gpu-support;
- установка TensorFlow без поддержки GPU — http://bit.ly/tensorflow-no-gpu.

Среди тех, кто не пользуется Linux, до того как компания Google опубликовала TensorFlow (http://deeplearning.net/software/theano/), привычным вариантом работы с матричной математикой с помощью GPU была библиотека Theano, активно разрабатываемая в Монреальском университете. Для нее создана страница, посвященная использованию GPU (http://deeplearning.net/software/theano/tutorial/using_gpu.html). Theano поддерживает операционные системы Windows, OS X и Linux. Доступна по команде `pip`:

```
$ pip install Theano
```

Для низкоуровневого взаимодействия с GPU вы можете использовать PyCUDA (https://developer.nvidia.com/pycuda).

Наконец, те, у кого нет NVIDIA GPU, могут использовать PyOpenCL (https://pypi.python.org/pypi/pyopencl), обертку для библиотеки OpenCL от Intel (https://software.intel.com/en-us/intel-opencl), которая совместима с несколькими разными аппаратными наборами (https://software.intel.com/en-us/articles/opencl-drivers).

Взаимодействие с библиотеками, написанными на C/C++/FORTRAN

Все библиотеки, описанные в следующих разделах, отличаются друг от друга. CFFI и ctypes написаны на Python, F2PY нужна для FORTRAN, SWIG позволяет использовать объекты языка C во многих языках (не только Python), а Boost.Python (библиотека C++) — объекты языка C++ в коде Python и наоборот. В табл. 8.2 приводится более подробная информация.

Таблица 8.2. Интерфейсы C и C++

Библиотека	Лицензия	Причины использовать
CFFI	Лицензия MIT	• Лучшая совместимость с PyPy. • Позволяет писать код C изнутри Python, который может быть скомпилирован для сборки общей библиотеки C со связыванием Python
ctypes	Лицензия Python Software Foundation	• Находится в стандартной библиотеке Python. • Позволяет оборачивать существующие DLL или общие объекты, которые писали не вы. • Вторая по качеству совместимость с PyPy
F2PY	Лицензия BSD	• Позволяет использовать библиотеку FORTRAN. • F2PY является частью библиотеки NumPy, поэтому вам следует применять NumPy
SWIG	GPL (output не ограничен)	Позволяет автоматически генерировать библиотеки на многих языках, используя специальный формат файла, который не похож ни на C, ни на Python
Boost.Python	Лицензия Boost Software	Этот инструмент не относится к командной строке. Это библиотека C++, которая может быть включена в код C++ и использована для определения того, какие объекты должны быть доступны Python

C Foreign Function Interface

Пакет CFFI (https://cffi.readthedocs.org/en/latest/) предоставляет простой механизм для взаимодействия с кодом, написанным на C, из Python и PyPy. CFFI рекомендуется использовать с PyPy (http://doc.pypy.org/en/latest/extending.html) для наилучшей

совместимости между CPython и PyPy. Он поддерживает два режима: встроенный режим совместимости для бинарных интерфейсов приложения (application binary interface, ABI) (смотрите следующий пример кода) позволяет динамически загружать и запускать функции из исполняемых модулей (по сути, предоставляет такую же функциональность, как LoadLibrary или dlopen), а также режим API, который позволяет выполнять сборку модулей расширения C[1].

Установите его с помощью `pip`:

```
$ pip install cffi
```

Рассмотрим пример взаимодействия ABI:

```
from cffi import FFI
ffi = FFI()
ffi.cdef("size_t strlen(const char*);") ❶
clib = ffi.dlopen(None)  ❷
length = clib.strlen("String to be evaluated.") ❸
# печатает 23
print("{}".format(length))
```

❶ Строка может быть получена из объявления функции, расположенного в заголовочном файле C.

❷ Открываем общую библиотеку (`*.DLL` или `*.so`).

❸ Теперь мы можем относиться к `clib` как к модулю Python и просто вызываем функции, которые определили с помощью точечной нотации.

ctypes

ctypes (https://docs.python.org/3/library/ctypes.html) — это выбор де-факто для взаимодействия с кодом на C/C++ и CPython; находится в стандартной библиотеке. Предоставляет полный доступ к нативному интерфейсу на C для большей части операционных систем (например, kernel32 для Windows, или libc для *nix), а также поддерживает загрузку и взаимодействие с динамическими библиотеками — разделяемыми объектами (`*.so`) или DLL — во время выполнения программы. Вместе с ctypes поставляется множество типов для взаимодействия с API системы, что позволяет вам легко определить собственные сложные типы вроде структур и объединений, а также модифицировать элементы вроде внутренних полей и выравнивания, если это нужно. Она может быть несколько неудобна в использовании (поскольку вам нужно вводить много дополнительных символов), но вместе с модулем стандартной библиотеки struct (https://docs.python.org/3.5/library/struct.html) у вас, по сути, будет полный контроль над тем, как

[1] Необходимо особенно тщательно писать расширения на C, чтобы убедиться, что вы регистрируете свои потоки для интерпретатора (http://docs.python.org/c-api/init.html#threads).

ваши типы данных преобразуются в типы, которые могут применять методы, написанные на чистом C/C++.

Например, структура C, определенная следующим образом в файле `my_struct.h`:

```
struct my_struct {
    int a;
    int b;
};
```

может быть реализована так, как показано в файле с именем my_struct.py:

```
import ctypes
class my_struct(ctypes.Structure):
    _fields_ = [("a", c_int),
                ("b", c_int)]
```

F2PY

Генератор интерфейса Fortran-к-Python (F2PY) (http://docs.scipy.org/doc/numpy/f2py/) является частью NumPy. Чтобы его получить, установите NumPy с помощью команды `pip`:

```
$ pip install numpy
```

Предоставляет гибкую функцию командной строки, `f2py`, которая может быть использована тремя разными способами (все они задокументированы в руководстве к F2PY по адресу http://docs.scipy.org/doc/numpy/f2py/getting-started.html). Если вы имеете доступ к управлению исходным кодом, то можете добавить особые комментарии с инструкциями для F2PY, которые показывают предназначение каждого аргумента (какие элементы являются входными, а какие — возвращаемыми), а затем запустить F2PY:

```
$ f2py -c fortran_code.f -m python_module_name
```

В противном случае F2PY способен сгенерировать промежуточный файл с расширением `*.pyf`, который вы можете модифицировать, чтобы получить такой же результат. Для этого потребуются три шага:

```
$ f2py fortran_code.f -m python_module_name -h interface_file.pyf ❶
$ vim interface_file.pyf ❷
$ f2py -c interface_file.pyf fortran_code.f ❸
```

❶ Автоматически сгенерируйте промежуточный файл, который определяет интерфейс между сигнатурами функций языков FORTRAN и Python.

❷ Отредактируйте файл, чтобы входные и выходные переменные были корректно размечены.

❸ Теперь скомпилируйте код и постройте модули расширения.

SWIG

Упрощенный упаковщик и генератор интерфейсов (Simplified Wrapper Interface Generator, SWIG) (http://www.swig.org/) поддерживает большое количество языков сценария, включая Python. Этот широко распространенный инструмент командной строки генерирует привязки для интерпретируемых языков из аннотированных файлов заголовков C/C++.

Для начала используйте SWIG для того, чтобы автоматически сгенерировать промежуточный файл из заголовка — он будет иметь суффикс `*.i`. Далее модифицируйте этот файл так, чтобы он отражал именно тот интерфейс, который вам нужен, а затем запустите инструмент сборки, чтобы скомпилировать код в разделяемую библиотеку. Все это описывается шаг за шагом в руководстве по SWIG (http://www.swig.org/tutorial.html).

Несмотря на то что есть некоторые ограничения (в данный момент могут иметься проблемы с небольшим объемом новой функциональности C++, а заставить работать код, содержащий большое количество шаблонов, может быть затруднительно), SWIG предоставляет много функций Python при малых усилиях. В дополнение вы легко можете расширить привязки, создаваемые SWIG (в файле интерфейса), чтобы перегрузить операторы и встроенные методы и, по сути, преобразовать исключения C++ таким образом, дабы вы могли отловить их в Python.

Рассмотрим пример, иллюстрирующий, как переопределить `__repr__`. Этот фрагмент кода получен из файла с именем `MyClass.h`:

```
#include <string>
class MyClass {
private:
    std::string name;
public:
    std::string getName();
};
```

А это `myclass.i`:

```
%include "string.i"
%module myclass
%{
#include <string>
#include "MyClass.h"
%}
%extend MyClass {
    std::string __repr__()
    {
        return $self->getName();
    }
}
%include "MyClass.h"
```

В репозитории SWIG на GitHub вы можете найти еще больше примеров использования Python (https://github.com/swig/swig/tree/master/Examples/python). Установите SWIG с помощью вашего менеджера пакетов, если он там есть (`apt-get install swig`, `yum install swig.i386` или `brew install swig`), или воспользуйтесь ссылкой http://www.swig.org/survey.html, чтобы загрузить SWIG, а затем следуйте инструкциям по установке для вашей операционной системы (http://www.swig.org/Doc3.0/Preface.html#Preface_installation). Если у вас нет библиотеки Perl Compatible Regular Expressions (PCRE) в OS X, для ее установки задействуйте Homebrew:

```
$ brew install pcre
```

Boost.Python

Boost.Python (http://www.boost.org/doc/libs/1_60_0/libs/python/doc/) требует выполнения несколько большего объема ручной работы для того, чтобы воспользоваться функциональностью объектов C++, но он может предложить ту же функциональность, что и SWIG, и даже больше, например обертку, позволяющую получать доступ к объектам Python как к объектам PyObjects в C++, а также инструменты для предоставления доступа к объектам C++ для кода Python. В отличие от SWIG, Boost.Python является библиотекой, а не инструментом командной строки, поэтому вам не нужно создавать промежуточный файл с другим форматированием — все пишется на языке C++. Для Boost.Python написано подробное руководство (http://bit.ly/boost-python-tutorial), если он вам интересен.

9 Программные интерфейсы

В этой главе мы сначала покажем, как применять Python для получения информации из API, которые используются для того, чтобы делиться данными между организациями. Затем мы опишем инструменты, которые многие организации, работающие с Python, применяют с целью поддержки коммуникации внутри своей инфраструктуры.

Мы уже рассмотрели поддержку конвейеров и очередей в Python между процессами в подразделе «Модуль multiprocessing» раздела «Скорость» главы 8. Для коммуникации *между компьютерами* требуется, чтобы обе стороны применяли заранее определенный набор протоколов: для Интернета применяется стек протоколов TCP/IP[1] (https://en.wikipedia.org/wiki/Internet_protocol_suite). Вы можете реализовать протокол UDP самостоятельно с помощью сокетов (https://pymotw.com/2/socket/udp.html), Python предлагает библиотеки ssl (предоставляет обертки TLS/SSL для сокетов) и asyncio (для реализации асинхронных транспортов для протоколов TCP, UDP, TLS/SSL) (https://docs.python.org/3/library/asyncio-protocol.html), а также конвейеры для подпроцессов.

---

[1] Стек протоколов TCP/IP (или Internet Protocol) имеет четыре концептуальные части: протоколы канального уровня указывают, как получить информацию от компьютера и Интернета. За их работу отвечают сетевые карты и операционные системы, но не программы Python. Протоколы сетевого уровня (IPv4, IPv6 и т. д.) управляют доставкой пакетов, состоящих из битов, от источника к месту назначения — стандартные варианты предоставлены в библиотеке сокетов для Python (https://docs.python.org/3/library/socket.html). Протоколы транспортного уровня (TCP, UDP и т. д.) указывают, как будут общаться две конечные точки. Возможные варианты также находятся в библиотеке сокетов. Наконец, протоколы прикладного уровня (FTP, HTTP и т. д.) указывают, как должны выглядеть данные для того, чтобы их могло использовать приложение (например, FTP применяется для передачи файлов, а HTTP — для передачи гипертекста) — в стандартной библиотеке Python предоставляются отдельные модули, реализующие наиболее распространенные протоколы.

Но большинство использует более высокоуровневые библиотеки, которые предоставляют клиенты, реализующие различные протоколы уровня приложения: `ftplib`, `poplib`, `imaplib`, `nntplib`, `smtplib`, `telnetlib` и `xmlrpc`. Все они предоставляют классы для обычных клиентов и клиентов, имеющих обертку TLS/SSL (urllib применяется для работы с запросами HTTP, но мы во многих случаях рекомендуем библиотеку Requests).

В первом разделе этой главы рассматриваются запросы HTTP: как получить данные из общедоступных API в Сети. Далее мы сделаем небольшое отступление и расскажем о сериализации в Python, а в третьем разделе опишем популярные инструменты для работы с сетями предприятий. Мы постараемся явно указывать, когда какой-то инструмент доступен только в Python 3. Если вы используете Python 2 и не можете найти модуль или класс, о котором мы говорим, рекомендуем взглянуть на этот список изменений между стандартными библиотеками Python 2 и Python 3: http://python3porting.com/stdlib.html.

Веб-клиенты

Протокол передачи гипертекста (Hypertext Transfer Protocol, HTTP) — это протокол приложения, предназначенный для распределенных, объединенных информационных систем, использующих гипермедиа. Является основным способом обмена данными во Всемирной сети. Данный раздел посвящен вопросу получения данных из Интернета с помощью библиотеки Requests.

Модуль стандартной библиотеки Python urllib предоставляет большую часть функциональности HTTP, которая вам может понадобиться, но на низком уровне для нее характерно выполнение немало объема работы для решения относительно простых задач (вроде получения данных от сервера HTTPS, который требует аутентификации). В документации к модулю `urllib.request` говорится, чтобы вы использовали вместо него библиотеку Requests.

Requests работает со всеми запросами HTTP в Python, что позволяет выполнить бесшовную интеграцию с веб-сервисами. Нет необходимости добавлять вручную строки запроса в ваши URL или выполнять кодирование ваших данных для команды POST. Поддержание соединений (устойчивые соединения HTTP) и объединение соединений HTTP в пулы доступны благодаря классу `request.sessions.Session`, поддерживаемому библиотекой urllib3 (https://pypi.python.org/pypi/urllib3), которая встроена в библиотеку Requests (это значит, что вам не нужно устанавливать ее отдельно). Вы можете получить ее с помощью `pip`:

```
$ pip install requests
```

В документации к Requests (http://docs.python-requests.org/en/latest/index.html) более подробно описывается все то, что мы будем рассматривать далее.

API для сети

Практически все — от Бюро переписи населения США до Национальной библиотеки Нидерландов — имеют API; его вы можете использовать для получения данных, которыми они хотят поделиться. Некоторые из этих API, вроде Twitter и Facebook, позволяют вам (или приложениям, которые вы используете) модифицировать эти данные. Возможно, вы слышали о термине RESTful API. REST расшифровывается как representational state transfer («передача состояния представления») — это парадигма, на которой основан способ проектирования HTTP 1.1, но она не является стандартом, протоколом или требованием. Однако большинство поставщиков API для веб-сервисов следуют принципам проектирования RESTful.

Воспользуемся кодом для того, чтобы проиллюстрировать распространенные термины:

```
import requests
                ❶                          ❷          ❸          ❹
result = requests.get('http://pypi.python.org/pypi/requests/json')
```

❶ *Метод* является частью протокола HTTP. В RESTful API разработчик выбирает, какое действие предпримет сервер, и указывает его вам в документации к API. По адресу http://bit.ly/http-method-defs содержится список всех методов, самые распространенные из них `GET`, `POST`, `PUT` и `DELETE`. Зачастую «глаголы HTTP» соответствуют своим именам — получают, изменяют или удаляют данные.

❷ *Базовый* URI является корнем API.

❸ Клиенты будут указывать конкретный *элемент*, для которого им нужны данные.

❹ Вы можете задать и другие *типы мультимедиа*.

Этот код выполнил запрос HTTP к ресурсу http://pypi.python.org/pypi/requests/json, который является бэкендом JSON для PyPI. Если вы взглянете на него в браузере, то увидите большую строку JSON. В библиотеке Requests возвращаемым значением для запроса HTTP будет объект типа `Response`:

```
>>> import requests
>>> response = requests.get('http://pypi.python.org/pypi/requests/json')
>>> type(response)
<class 'requests.models.Response'>
>>> response.ok
True
>>> response.text    # Эта команда возвращает весь текст ответа
>>> response.json()  # Эта команда преобразует текст ответа в словарь
```

PyPI вернул текст в формате JSON. Не существует правила, которое регулирует формат отправляемых данных, но во многих API используются JSON или XML.

Анализ JSON. Нотация объектов Javascript (Javascript Object Notation, JSON) полностью соответствует своему имени: используется для определения объектов в JavaScript. Библиотека Requests имеет встроенный анализатор JSON в объектах типа `Response`.

Библиотека json (https://docs.python.org/3/library/json.html) может анализировать JSON, расположенный в строках или файлах, и помещать его в словарь Python (или список, если это вам так удобнее). Она также преобразует словари или списки Python в строки JSON. Например, в следующей строке содержатся данные JSON:

```
json_string = '{"first_name": "Guido", "last_name":"van Rossum"}'
```

Проанализировать ее можно следующим образом:

```
import json
parsed_json = json.loads(json_string)
```

Теперь вы можете использовать эти данные как словарь:

```
print(parsed_json['first_name'])
"Guido"
```

Вы также можете преобразовать следующий словарь в JSON:

```
d = {
    'first_name': 'Guido',
    'last_name': 'van Rossum',
    'titles': ['BDFL', 'Developer'],
}
print(json.dumps(d))
'{"first_name": "Guido", "last_name": "van Rossum",
  "titles": ["BDFL", "Developer"]}'
```

simplejson для ранних версий Python

Библиотека json была добавлена в Python 2.6. Если вы используете более раннюю версию Python, можете загрузить из PyPI библиотеку simplejson (https://simplejson.readthedocs.org/en/latest/).

simplejson предоставляет такой же API, как и модуль стандартной библиотеки json, но обновляется чаще, чем Python. Разработчики, работающие с более старыми версиями Python, все еще могут использовать функциональность, доступную в библиотеке json, импортировав simplejson. Вы можете выбрать simplejson как полноценную замену для json следующим образом:

```
import simplejson as json
```

После импортирования библиотеки simplejson под именем json все предыдущие примеры будут работать так же, как если бы вы использовали стандартную библиотеку json.

Анализ XML

В стандартной библиотеке есть анализатор XML (методы `parse()` и `fromstring()` класса `xml.etree.ElementTree`), но `this` использует библиотеку Expat и создает объект `ElementTree`, сохраняющий структуру XML. Это значит, что требуется итерировать по нему и опрашивать потомков, дабы получить содержимое. Если нужно лишь получить данные, обратитесь к `untangle` или `xmltodict`. Вы можете установить их командой `pip`:

```
$ pip install untangle
$ pip install xmltodict
```

- untangle (https://github.com/stchris/untangle). Принимает документ XML и возвращает объект Python, чья структура отражает узлы и атрибуты документа. Например, такой файл XML:

  ```
  <?xml version="1.0" encoding="UTF-8"?>
  <root>
      <child name="child1" />
  </root>
  ```

  можно загрузить следующим образом:

  ```
  import untangle
  obj = untangle.parse('path/to/file.xml')
  ```

  можно получить имя элемента-потомка:

  ```
  obj.root.child['name']  # is 'child1'
  ```

- xmltodict (http://github.com/martinblech/xmltodict). Преобразует XML в словарь. Например, такой файл XML:

  ```
  <mydocument has="an attribute">
    <and>
      <many>elements</many>
      <many>more elements</many>
    </and>
    <plus a="complex">
      element as well
    </plus>
  </mydocument>
  ```

  можно загрузить в объект типа `OrderedDict` (из модуля collections стандартной библиотеки Python):

  ```
  import xmltodict
  with open('path/to/file.xml') as fd: doc = xmltodict.parse(fd.read())
  ```

  можно получить доступ к элементам, атрибутам и значениям:

  ```
  doc['mydocument']['@has']  # is u'an attribute'
  doc['mydocument']['and']['many']  # is [u'elements', u'more elements']
  doc['mydocument']['plus']['@a']  # is u'complex'
  doc['mydocument']['plus']['#text']  # is u'element as well'
  ```

  С помощью xmltodict можно преобразовать словарь обратно в XML, вызвав функцию `unparse()`. Она имеет потоковый режим, подходящий для обработки файлов, не помещающихся в память, а также поддерживает пространства имен.

Скраппинг сайтов

Сайты не всегда предлагают данные в удобных форматах вроде CSV или JSON, но HTML представляет собой структурированные данные — здесь вступает в дело скраппинг.

Скраппинг сайтов — это использование компьютерной программы для анализа веб-страницы и сбора необходимых данных в формате, наиболее удобном для вас (при этом сохраняя их структуру).

По мере того как сайты предлагают свои API, они явно просят вас не использовать скраппинг: API открывает вам доступ только к тем данным, которыми владельцы сайта желают поделиться. Перед тем как начать скраппинг, прочтите условия использования целевого сайта и будьте законопослушным гражданином.

lxml

lxml (http://lxml.de/) — это довольно обширная библиотека, написанная для выполнения быстрого анализа документов XML и HTML. Позволяет обрабатывать некоторый объем некорректной разметки.

Загрузите ее с помощью `pip`:

```
$ pip install lxml
```

Используйте метод `requests.get`, чтобы получить веб-страницу с данными, преобразуйте их с помощью модуля html и сохраните результат в дереве:

```
from lxml import html
import requests

page = requests.get('http://econpy.pythonanywhere.com/ex/001.html')  ❶
tree = html.fromstring(page.content)  ❷
```

❶ Это реальная веб-страница, и данные, которые мы показываем, тоже реальные (вы можете посетить эту страницу в браузере).

❷ Мы используем свойство `page.content`, а не `page.text`, поскольку метод `html.fromstring()` неявно ожидает получить объект типа `bytes`.

Теперь дерево содержит весь файл HTML и имеет удобную структуру. Мы можем пойти двумя путями: использовать XPath (http://lxml.de/xpathxslt.html) или CSSSelect (http://lxml.de/cssselect.html). Оба этих способа стандартные для указания пути с помощью дерева HTML, они определены и поддерживаются World Wide Web Consortium (W3C) и реализованы как модули в lxml. В этом примере мы используем XPath. Руководство по XPath (http://www.w3schools.com/xsl/xpath_intro.asp) поможет вам начать работу.

Существуют различные инструменты для получения XPath элементов изнутри вашего браузера вроде Firebug for Firefox или Chrome Inspector. Если используете Chrome, щелкните правой кнопкой мыши на элементе, выберите пункт меню Inspect element (Инспектировать элемент), подсветите код, снова щелкните правой кнопкой и выберите Copy XPath (Скопировать XPath).

После небольшого анализа мы видим, что данные на нашей странице содержатся в двух элементах: `div` (с заголовком `buyer-name`) и `span` (имеющий класс `item-price`):

```
<div title="buyer-name">Carson Busses</div>
<span class="item-price">$29.95</span>
```

Зная это, мы можем создать корректный запрос XPath и использовать lxml-функцию `xpath`, как показано в примере:

```
# Это создаст список покупателей:
buyers = tree.xpath('//div[@title="buyer-name"]/text()')
# Это создаст список цен
prices = tree.xpath('//span[@class="item-price"]/text()')
```

Посмотрим, что получилось:

```
>>> print('Buyers: ', buyers)
Buyers:  ['Carson Busses', 'Earl E. Byrd', 'Patty Cakes',
'Derri Anne Connecticut', 'Moe Dess', 'Leda Doggslife', 'Dan Druff',
'Al Fresco', 'Ido Hoe', 'Howie Kisses', 'Len Lease', 'Phil Meup',
'Ira Pent', 'Ben D. Rules', 'Ave Sectomy', 'Gary Shattire',
'Bobbi Soks', 'Sheila Takya', 'Rose Tattoo', 'Moe Tell']
>>>
>>> print('Prices: ', prices)
Prices:  ['$29.95', '$8.37', '$15.26', '$19.25', '$19.25',
'$13.99', '$31.57', '$8.49', '$14.47', '$15.86', '$11.11',
'$15.98', '$16.27', '$7.50', '$50.85', '$14.26', '$5.68',
'$15.00', '$114.07', '$10.09']
```

Сериализация данных

Сериализация данных — это преобразование структурированных данных в формат, который позволяет делиться ими или сохранить, при этом вы можете воссоздать объект в памяти на получающей стороне (или при чтении из хранилища). В некоторых случаях еще одной причиной сериализации данных является минимизация сериализованных данных, что в свою очередь минимизирует занятое дисковое пространство или требования к полосе пропускания.

В следующих разделах рассматриваются формат Pickle, характерный для Python, некоторые инструменты сериализации между языками, способы сжатия, предлагаемые стандартной библиотекой Python, а также протокол буфера, который может снизить количество операций копирования данных перед их передачей.

Pickle

Нативный модуль сериализации данных для Python называется Pickle (https://docs.python.org/2/library/pickle.html). Рассмотрим пример его использования:

```
import pickle
# Пример словаря
grades = { 'Alice': 89, 'Bob': 72, 'Charles': 87 }
# Используем дампы для преобразования объекта в сериализованную строку
serial_grades = pickle.dumps( grades )
# Используем loads для десериализации строки в объект
received_grades = pickle.loads( serial_grades )
```

Функции, методы, классы и эфемерные объекты вроде конвейеров сериализовать нельзя.

В соответствии с документацией к Pickle «модуль pickle небезопасно использовать для ошибочных или вредоносных данных. Никогда не десериализуйте данные, полученные из недостоверных источников».

Межъязыковая сериализация

Если вы ищете модуль сериализации данных, который поддерживает несколько языков, подойдут Protobuf от Google (https://developers.google.com/protocol-buffers/docs/pythontutorial) и Avro от Apache (https://avro.apache.org/docs/1.7.6/gettingstartedpython.html).

В стандартной библиотеке имеется библиотека xdrlib (https://docs.python.org/3/library/xdrlib.html), позволяющая упаковывать и распаковывать данные в формате External Data Representation (XDR) (https://en.wikipedia.org/wiki/External_Data_Representation) от компании Sun. Этот формат не зависит от операционной системы и протокола передачи данных. Он работает на гораздо более низком уровне, нежели предыдущие варианты, и просто выполняет конкатенацию упакованных байтов, поэтому и клиент, и сервер должны знать тип и порядок упаковки. Рассмотрим пример сервера, получающего данные в формате XDR:

```
import socketserver
import xdrlib

class XdrHandler(socketserver.BaseRequestHandler):
    def handle(self):
        data = self.request.recv(4)  ❶
        unpacker = xdrlib.Unpacker(data)
        message_size = self.unpacker.unpack_uint()  ❷
        data = self.request.recv(message_size)  ❸
        unpacker.reset(data)  ❹
        print(unpacker.unpack_string())  ❺
        print(unpacker.unpack_float())
```

```
import socketserver
import xdrlib

class XdrHandler(socketserver.BaseRequestHandler):
    def handle(self):
        data = self.request.recv(4)  ❶
        unpacker = xdrlib.Unpacker(data)
        message_size = self.unpacker.unpack_uint()  ❷
        data = self.request.recv(message_size)  ❸
        unpacker.reset(data)  ❹
        print(unpacker.unpack_string())  ❺
        print(unpacker.unpack_float())
        self.request.sendall(b'ok')

server = socketserver.TCPServer(('localhost', 12345), XdrHandler)
server.serve_forever()
```

❶ Данные могут иметь произвольную длину, поэтому мы добавили в начало файла упакованное беззнаковое число (4 байта), содержащее размер сообщения.

❷ Мы должны знать заранее, что получаем данные типа `unsigned int`.

❸ В этой строке считываем остальную часть сообщения...

❹ ...а в этой сбрасываем распаковщик, чтобы он начал работать с новыми данными.

❺ Мы должны знать заранее, что получаем одну строку, а затем одно число с плавающей точкой.

Конечно, если обе стороны являлись программами Python, вы бы использовали Pickles. Но если сервер написан на каком-то другом языке, то соответствующий код клиента, отправляющего данные, выглядел бы так:

```
import socket
import xdrlib

p = xdrlib.Packer()
p.pack_string('Thanks for all the fish!') ❶
p.pack_float(42.00)
xdr_data = p.get_buffer()
message_length = len(xdr_data)

p.reset() ❷
p.pack_uint(message_length)
len_plus_data = p.get_buffer() + xdr_data  ❸

with socket.socket() as s:
    s.connect(('localhost', 12345))
    s.sendall(len_plus_data)
    if s.recv(1024):
      print('success')
```

❶ Сначала упакуем все данные, подлежащие отправке.

❷ Далее отдельно упакуем длину сообщения...

❸ ...и добавим ее ко всему сообщению.

Сжатие

Стандартная библиотека Python поддерживает сжатие и декомпрессию данных с использованием алгоритмов zlib, gzip, bzip2 и lzma, а также создание архивов ZIP и TAR. Для того чтобы поместить в ZIP-архив данные, сериализованные с помощью Pickle, сделайте следующее:

```
import pickle
import gzip
data = "my very big object"
# Для запаковки и сериализации:
with gzip.open('spam.zip', 'wb') as my_zip:
    pickle.dump(data, my_zip)
# Для распаковки и десериализации:
with gzip.open('spam.zip', 'rb') as my_zip:
    unpickled_data = pickle.load(my_zip)
```

Протокол буфера

Элай Бендерски (Eli Bendersky), один из основных разработчиков Python, написал статью, посвященную вопросу снижения количества копий одних и тех же данных, хранящихся в памяти, с помощью буферов памяти (http://tinyurl.com/bendersky-buffer-protocol). Используя этот прием, вы даже можете считать данные из файла или сокета и поместить их в существующий буфер. Для получения более подробной информации обратитесь к документации для буферов протоколов и PEP 3118 (https://docs.python.org/3/c-api/buffer.html), где предлагаются улучшения, которые были реализованы в Python 3 и обратно портированы для версий Python 2.6 и выше.

Распределенные системы

Распределенные вычислительные системы выполняют задачу сообща (вроде игр, чат-комнат в Интернете или расчетов Hadoop) путем передачи информации друг другу.

В этом разделе сначала показываются самые популярные библиотеки для выполнения распространенных задач, связанных с сетью, а далее рассматривается шифрование (эта тема неотрывно следует за темой работы с сетью).

Работа с сетью

В Python коммуникация для соединенных сетей зачастую обрабатывается с помощью асинхронных инструментов или потоков, что позволяет обойти ограничение в один поток, создаваемое глобальной блокировкой интерпретатора. Все библиотеки, перечисленные в табл. 9.1, решают одну и ту же проблему — обходят GIL — с помощью разной функциональности.

Таблица 9.1. Работа с сетью

Библиотека	Лицензия	Причины использовать
asyncio	Лицензия PSF	• Предоставляет асинхронный цикл событий для управления коммуникацией с помощью неблокирующих сокетов и очередей, а также сопрограммами, определяемыми пользователем. • Содержит асинхронные сокеты и очереди
gevent	Лицензия MIT	• Тесно связана с libev — библиотекой для асинхронного ввода/вывода, написанной на C. • Предоставляет быстрый сервер WSGI, созданный на основе сервера HTTP. • Содержит полезный модуль gevent.monkey (http://www.gevent.org/gevent.monkey.html), который имеет функции корректировки для стандартной библиотеки, что позволяет применять сторонние модули, использующие блокирующие сокеты
Twisted	Лицензия MIT	• Предоставляет асинхронные реализации более новых протоколов, например GPS, Internet of Connected Products (IoCP), и протокола Memcached (https://memcached.org/). • В ее цикл событий интегрированы разнообразные фреймворки, управляемые событиями вроде wxPython или GTK. • Имеет встроенный сервер SSH и клиентские инструменты
PyZMQ	Лицензии LGPL (ZMQ) и BSD (часть с Python)	• Позволяет настроить неблокирующие очереди сообщений, использующие API, похожий на сокеты, а также взаимодействовать с ними. • Предоставляет поведения сокетов (запрос/ответ, публикация/подписка и отправка/получение), которые поддерживают распределенные вычисления. • Используйте эту библиотеку, если хотите создать собственную инфраструктуру для коммуникации; в ее имени содержится буква Q, но она не похожа на RabbitMQ — ее можно использовать для того, чтобы создать что-то вроде RabbitMQ или что-то, что имеет совершенно другое поведение (в зависимости от выбранных шаблонов сокетов)

Библиотека	Лицензия	Причины использовать
pika	Лицензия BSD	• Предоставляет легковесный клиент AMQP (протокол коммуникации) для соединения с RabbitMQ или другими брокерами сообщений. • Включает в себя адаптеры, подходящие для использования в циклах событий Tornado или Twisted. • Используйте ее вместе с брокером сообщений вроде RabbitMQ, если вам нужна более легковесная библиотека (без информационных панелей и других свистелок), которая позволяет отправлять содержимое внешнему брокеру сообщений вроде RabbitMQ
Celery	Лицензия BSD	• Предоставляет клиент AMQP для соединения с RabbitMQ или другими брокерами сообщений. • Дает возможность сохранять состояния задач в бэкенде, который может использовать другие популярные варианты вроде соединения с базой данных с помощью SQLAlchemy, Memcached или другим способом. • Имеет необязательный к использованию инструмент для веб-администрирования и наблюдения, который называется Flower. • Может быть использована вместе с брокером сообщений вроде RabbitMQ в качестве моментально готовой к использованию системы-брокера сообщений

Производительность сетевых инструментов из стандартной библиотеки Python

Инструмент asyncio (https://docs.python.org/3/library/asyncio.html) был представлен в Python 3.4. Включает в себя идеи, почерпнутые у сообществ разработчиков вроде тех, что поддерживают библиотеки Twisted и gevent. Это инструмент для работы с конкуренцией, а самым частым приложением конкуренции являются сетевые сервера. В документации к asyncore (предшественнике asyncio) говорится следующее:

> Существует лишь два способа заставить программу, работающую на одном процессоре, выполнять «больше одной задачи одновременно». Многопоточное программирование — самый простой и популярный способ сделать это, но существует еще один прием, который позволяет воспользоваться практически всеми преимуществами многопоточности, не задействуя на самом деле более одного потока. Применять этот прием имеет смысл, только если ваша программа ограничена по вводу/выводу. Если программа ограничена по процессору, то заранее запланированные потоки — это, возможно, именно то, что вам нужно. Однако сетевые сервера редко бывают ограниченными по процессору.

asyncio все еще находится в стандартной библиотеке Python на временной основе — ее API может измениться и потерять обратную совместимость, поэтому сильно не привыкайте.

Не вся функциональность нова — asyncore (объявлена устаревшей в Python 3.4) имеет цикл событий, асинхронные сокеты[1] и асинхронный ввод/вывод информации из файлов, а asynchat (также объявлена устаревшей в Python 3.4) имеет асинхронные очереди[2]. В asyncio добавлен один важный элемент — формализованная реализация *сопрограмм*. В Python это формально определяется как *функция сопрограммы*, то есть функция, чье описание начинается с конструкции `async def`, а не просто с `def` (если используется старый синтаксис, то применяется декоратор `@asyncio.coroutine`), и как объект, получаемый путем вызова функции сопрограммы (некого рода вычислений или операций ввода/вывода). Сопрограмма может обращаться к процессору и получить возможность участвовать в асинхронном цикле событий по очереди вместе с другими сопрограммами.

Множество страниц документации посвящено примерам, помогающим сообществу, поскольку такая концепция для языка новая. Она прозрачна, продуманна, и на нее определенно стоит обратить внимание. В этой интерактивной сессии мы просто хотим показать функции для цикла событий и некоторые доступные классы:

```
>>> import asyncio
>>>
>>> [l for l in asyncio.__all__ if 'loop' in l]
['get_event_loop_policy', 'set_event_loop_policy',
'get_event_loop', 'set_event_loop', 'new_event_loop']
>>>
>>> [t for t in asyncio.__all__ if t.endswith('Transport')]
['BaseTransport', 'ReadTransport', 'WriteTransport', 'Transport',
'DatagramTransport', 'SubprocessTransport']
>>>
>>> [p for p in asyncio.__all__ if p.endswith('Protocol')]
['BaseProtocol', 'Protocol', 'DatagramProtocol',
'SubprocessProtocol', 'StreamReaderProtocol']
>>>
>>> [q for q in asyncio.__all__ if 'Queue' in q]
['Queue', 'PriorityQueue', 'LifoQueue', 'JoinableQueue',
'QueueFull', 'QueueEmpty']
```

---

[1] Сокет состоит из трех элементов: IP-адреса, включая номер порта, транспортного протокола (вроде TCP/UDP) и канала ввода/вывода (объект, похожий на файл). В документации к Python приводится отличное введение в тему сокетов (https://docs.python.org/3/howto/sockets.html).

[2] Для очереди не требуются IP-адрес или протокол, поскольку она реализуется на одном компьютере; вы просто записываете в нее данные — и другой процесс может их прочитать. Очередь похожа на multiprocessing.Queue, но здесь операции ввода/вывода выполняются асинхронно.

gevent

gevent (http://www.gevent.org/) — это библиотека Python для работы с сетью, основанная на сопрограммах. Использует гринлеты, чтобы предоставить высокоуровневый синхронный API на базе цикла событий библиотеки libev (http://software.schmorp.de/pkg/libev.html), написанной на C. Гринлеты основаны на библиотеке greenlet (http://greenlet.readthedocs.io/en/latest/) — миниатюрные зеленые потоки (https://en.wikipedia.org/wiki/Green_threads) (или потоки уровня пользователя, по смыслу противоположные потоками, управляемым ядром), которые разработчик может свободно заморозить, переключаясь между гринлетами. Если хотите получить более подробную информацию, обратите внимание на семинар Кавьи Джоши (Kavya Joshi) A Tale of Concurrency Through Creativity in Python (http://bit.ly/kavya-joshi-seminar).

Многие пользуются gevent, поскольку она легковесна и тесно связана с лежащей в ее основе библиотекой libev, написанной на C, что повышает производительность. Если вам нравится идея интеграции асинхронного ввода/вывода и гринлетов, эта библиотека отлично вам подойдет. Установите ее с помощью pip:

```
$ pip install gevent
```

Рассмотрим пример из документации к greenlet:

```
>>> import gevent
>>>
>>> from gevent import socket
>>> urls = ['www.google.com', 'www.example.com', 'www.python.org']
>>> jobs = [gevent.spawn(socket.gethostbyname, url) for url in urls]
>>> gevent.joinall(jobs, timeout=2)
>>> [job.value for job in jobs]
['74.125.79.106', '208.77.188.166', '82.94.164.162']
```

В документации содержится множество других примеров.

Twisted

Twisted (http://twistedmatrix.com/trac/) — это управляемый событиями движок для работы с сетями. Он может применяться для создания приложений на основе разных сетевых протоколов, включая серверы и клиенты HTTP, а также приложений, использующих протоколы SMTP, POP3, IMAP или SSH, протоколы мгновенного обмена сообщениями, и многих других (http://twistedmatrix.com/trac/wiki/Documentation). Установите его с помощью команды `pip`:

```
$ pip install twisted
```

Twisted существует с 2002 года и имеет верное сообщество. Ее можно назвать Emacs среди библиотек сопрограмм: все функции встроены (поскольку функциональность должна быть асинхронной для того, чтобы вы могли работать). Возможно, наиболее полезными инструментами являются асинхронная оболочка для соединений с базой

данных (расположен в `twisted.enterprise.adbapi`), DNS-сервер (в `twisted.names`), прямой доступ к пакетам (в `twisted.pair`) и дополнительные протоколы вроде AMP, GPS и SOCKSv4 (в `twisted.protocols`). Большая часть функциональности Twisted работает и в Python 3. Когда вы вызываете команду `pip install` в среде Python 3, вы получаете все библиотеки, портированные к этому моменту. Если вы нашли то, что вам нужно, в API (http://twistedmatrix.com/documents/current/api/moduleIndex.html), которого нет в вашей версии Twisted, вам стоит воспользоваться Python 2.7.

Для получения более подробной информации см. книгу Twisted (издательство O'Reilly) Джессики МакКеллар (Jessica McKellar) и Эйба Феттига (Abe Fettig). В дополнение к ней по адресу http://twistedmatrix.com/documents/current/core/examples/ приводится более 42 примеров использования Twisted, а в этом показываются их недавние достижения в скорости (http://speed.twistedmatrix.com/).

PyZMQ

PyZMQ (http://zeromq.github.com/pyzmq/) — это привязка к Python для ZeroMQ (http://www.zeromq.org/). Вы можете установить ее с помощью команды `pip`:

```
$ pip install pyzmq
```

ØMQ (также записывается как ZeroMQ, 0MQ или ZMQ) — библиотека для обмена сообщениями, которая имеет API, похожий на API сокетов. Предназначена для использования в масштабируемых распределенных или одновременно выполняемых приложениях. По сути, она реализует асинхронные сокеты и очереди, а также предоставляет пользовательский список «типов» сокетов, которые определяют, как работает ввод/вывод для каждого сокета. Рассмотрим пример:

```
import zmq
context = zmq.Context()
server = context.socket(zmq.REP)  ❶
server.bind('tcp://127.0.0.1:5000')  ❷

while True:
    message = server.recv().decode('utf-8')
    print('Client said: {}'.format(message))
    server.send(bytes('I don't know.', 'utf-8'))

# ~~~~~ и в другом файле ~~~~~

import zmq
context = zmq.Context()
client = context.socket(zmq.REQ)  ❸
client.connect('tcp://127.0.0.1:5000')  ❹

client.send(bytes("What's for lunch?", 'utf-8'))
response = client.recv().decode('utf-8')
print('Server replied: {}'.format(response))
```

❶ Тип сокета `zmq.REP` соответствует ее парадигме «запрос-ответ».

❷ Как и в случае обычных сокетов, вы привязываете сервер к IP-адресу и порту.

❸ Клиент имеет тип `zmq.REQ` — ZMQ определяет следующие константы: `zmq.REQ`, `zmq.REP`, `zmq.PUB`, `zmq.SUB`, `zmq.PUSH`, `zmq.PULL`, `zmq.PAIR`. Они устанавливают порядок отправки и принятия данных сокетом.

❹ Как и обычно, клиент соединяется с IP-адресом и портом, привязанным к серверу.

Эта реализация выглядит и «крякает» как сокеты, улучшенные с помощью добавления очередей и разнообразных шаблонов ввода/вывода.

Идея использования шаблонов заключается в том, чтобы предоставить строительный материал для распределенной сети.

Сокеты имеют следующие основные шаблоны.

- *Запрос — ответ*. `zmq.REQ` и `zmq.REP` соединяют набор клиентов с набором сервисов. Это может использоваться для создания шаблонов удаленного вызова процедуры или распределения задач.
- *Публикация — подписка*. `zmq.PUB` и `zmq.SUB` соединяют набор публикаторов с набором подписчиков. Этот шаблон предназначен для распространения данных (один узел распространяет данные другим), также с его помощью можно создать дерево распределения.
- *Отправка — получение (или конвейер)*. `zmq.PUSH` и `zmq.PULL` соединяют узлы с помощью шаблона разветвления на входе и выходе, который может иметь несколько шагов, а также циклы. С помощью этого шаблона реализуется параллельное распределение и сбор задач.

Одно из основных преимуществ ZeroMQ перед ориентированным на работу с сообщениями промежуточным ПО — библиотеку можно использовать для размещения сообщений в очереди без привлечения выделенного брокера сообщений. В документации к PyZMQ (http://pyzmq.readthedocs.io/) указаны новые возможности вроде туннелирования с помощью SSH. Остальную часть документации к ZeroMQ API лучше искать в основном руководстве к ZeroMQ (http://zguide.zeromq.org/page:all).

RabbitMQ

RabbitMQ (http://www.rabbitmq.com/) — это брокер сообщений с открытым исходным кодом, реализующий протокол Advanced Message Queuing Protocol (AMQP). Брокер сообщений — промежуточная программа, которая получает сообщения с отправляющей стороны и пересылает их получателям в соответствии с протоколом. Любой клиент, который реализует AMQP, может связываться с RabbitMQ. Для того чтобы получить RabbitMQ, перейдите на его страницу загрузки (https://www.rabbitmq.com/download.html) и следуйте инструкциям для вашей операционной системы.

Клиентские библиотеки, взаимодействующие с брокером, доступны для всех крупных языков программирования. Основными двумя библиотеками для Python являются pika и Celery — их можно установить с помощью `pip`:

```
$ pip install pika
$ pip install celery
```

- *pika* (https://pypi.python.org/pypi/pika). Это легковесный клиент AMQP 0-9-1, написанный на чистом Python. Считается предпочтительным для RabbitMQ. В руководствах по RabbitMQ (https://www.rabbitmq.com/getstarted.html) используется именно pika. Кроме того, целая страница https://pika.readthedocs.io/en/0.10.0/examples.html посвящена примерам работы с pika. Мы рекомендуем поработать с pika, как только вы установите RabbitMQ, независимо от того, какую библиотеку вы в итоге решите использовать, поскольку она довольно прямолинейна без дополнительной функциональности (это делает ее концепт довольно понятным).
- *Celery* (https://pypi.python.org/pypi/celery). Это клиент AMQP, который имеет гораздо больше функциональности: может использовать в качестве брокера сообщений RabbitMQ или Redis (распределенное хранилище данных в оперативной памяти), а также отслеживать задачи и результаты (и опционально сохранять их в выбранном пользователем бэкенде). Этот клиент имеет инструмент Flower (https://pypi.python.org/pypi/flower) для веб-администрирования/наблюдения за задачами. Он популярен среди веб-разработчиков. Существуют интеграционные пакеты для Django, Pyramid, Pylons, web2py и Tornado (для Flask такой пакет не нужен). Начинать работу лучше с руководства к Celery (http://tinyurl.com/celery-first-steps).

Шифрование

В 2013 году сформировалась Python Cryptographic Authority (PyCA) (https://github.com/pyca) — группа разработчиков, заинтересованных в создании высококачественных библиотек для шифрования[1]. Они предоставляют инструменты для шифрования и дешифрования сообщений на основе соответствующих ключей, а также криптографические хэш-функции, предназначенные для необратимого, но постоянного обфусцирования паролей или других секретных данных.

[1] Рождение библиотеки cryptography, а также история ее создания описаны в статье Джейка Эджа (Jake Edge) The state of crypto in Python (http://bit.ly/raim-kehrer-talk). cryptography — это низкоуровневая библиотека, предназначенная для импортирования более высокоуровневыми библиотеками вроде pyOpenSSL. Эдж цитирует беседу Джаррета Рэйма (Jarret Raim) и Пола Керера (Paul Kehrer), посвященную State of Crypto in Python (https://www.youtube.com/watch?v=r_Pj__qjBvA), и утверждает, что их набор тестов состоит более чем из 66 000 элементов, которые запускаются 77 раз при каждой сборке.

За исключением pyCrypto все библиотеки, представленные в табл. 9.2, поддерживаются PyCA. Практически все они созданы на основе библиотеки OpenSSL (https://www.openssl.org/), написанной на C (кроме тех, где это указано).

Таблица 9.2. Реализация шифрования

Реализация	Лицензия	Причины использовать
ssl и hashlib (а в Python 3.6 еще и secrets)	Лицензия Python Software Foundation	• Hashlib предоставляет неплохой алгоритм хэширования паролей, обновляемый по мере выхода новых версий Python, а ssl предоставляет клиент SSL/TLS (и сервер, но для него, возможно, обновления будут не самыми свежими). • Secrets — это генератор случайных чисел. Подходит для использования в криптографических целях
pyOpenSSL	Лицензия Apache v2.0	Использует самую последнюю версию OpenSSL в Python и предоставляет функции OpenSSL, недоступные в модуле стандартной библиотеки ssl
PyNaCl	Лицензия Apache v2.0	Содержит привязки к Python для libsodium*
libnacl	Лицензия Apache	Представляет собой интерфейс Python для libsodium для тех, кто использует стек Salt (http://saltstack.com/)
cryptography	Лицензия Apache v2.0 или BSD	Предоставляет прямой доступ к криптографическим примитивам, созданным на основе OpenSSL. Большинство пользователей предпочитают более высокоуровневый pyOpenSSL
pyCrypto	Открытый доступ	Эта библиотека старше остальных, она создана на основе собственной библиотеки, написанной на C. Раньше была частью наиболее популярной библиотеки для работы с шифрованием для Python
bcrypt	Лицензия Apache v2.0	Предоставляет хэш-функцию bcrypt**. Полезна для тех, кто хочет использовать py-bcrypt (или работал с ней ранее)

* libsodium (https://download.libsodium.org/doc/) — это версия библиотеки Networking and Cryptography (NaCl, произносится salt — соль); ее философия заключается в том, чтобы следовать определенным алгоритмам, которые имеют высокую производительность и которые легко использовать.

** Библиотека на самом деле содержит исходный код, написанный на C, и выполняет его сборку во время установки с помощью интерфейса C Fast Function, который мы описывали ранее. Bcrypt (https://en.wikipedia.org/wiki/Bcrypt) основан на алгоритме шифрования Blowfish.

В следующих разделах приведена дополнительная информация о библиотеках из табл. 9.2.

ssl, hashlib и secrets

Модуль ssl (https://docs.python.org/3/library/ssl.html) стандартной библиотеки Python предоставляет API для сокетов (`ssl.socket`), который ведет себя как обычный сокет, обернутый в протокол SSL, а также `ssl.SSLContext`, содержащий конфигурации для соединения по SSL. http (или httplib в Python 2) использует его для поддержки HTTPS. Если вы работаете с Python 3.5, можете задействовать memory BIO (https://docs.python.org/3/whatsnew/3.5.html#ssl) — сокет будет записывать входную/выходную информацию в буфер, а не в место назначения, что позволяет, например, кодировать/декодировать данные в шестнадцатеричный формат перед записью или при чтении.

Основная часть улучшений безопасности появилась в Python 3.4 — вы можете узнать больше из заметок о выпуске (https://docs.python.org/3.4/whatsnew/3.4.html) (появилась поддержка транспортных протоколов и алгоритмов хэширования). Это оказалось настолько важно, что функциональность была портирована в Python 2.7 (описывается в PEP 466 и PEP 476). Вы можете узнать больше сведений из речи Бенджамина Питерсона (Benjamin Peterson) о состоянии ssl в Python (http://bit.ly/peterson-talk).

Если вы работаете с Python 2.7, убедитесь, что у вас установлена версия 2.7.9 или выше или что для вашей версии используется PEP 476, — по умолчанию клиенты HTTP будут выполнять проверку сертификата при соединении с помощью протокола https. Либо всегда используйте библиотеку Requests (является вариантом по умолчанию).

Команда разработчиков Python рекомендует использовать значения по умолчанию для SSL, если ваша политика безопасности не предъявляет особых требований к клиенту. В этом примере показывается безопасный почтовый клиент (вы можете найти этот код в документации в библиотеке ssl по адресу http://bit.ly/ssl-security-consider (раздел Security considerations)):

```
>>> import ssl, smtplib
>>> smtp = smtplib.SMTP("mail.python.org", port=587)
>>> context = ssl.create_default_context()
>>> smtp.starttls(context=context)
(220, b'2.0.0 Ready to start TLS')
```

Для того чтобы убедиться в том, что сообщение не было повреждено во время передачи, используйте модуль hmac, который реализует алгоритм Keyed-Hashing for Message Authentication (HMAC), описанный RFC 2104 (https://tools.ietf.org/html/rfc2104.html). Он работает с сообщениями, хэшированными с помощью любого алгоритма из множества `hashlib.algorithms_available`. Для получения более подробной информации обратитесь к примеру из статьи Python Module of the Week (https://pymotw.com/2/hmac/). Если модуль у вас установлен, метод `hmac.compare_digest()`

позволяет выполнять любые криптографические алгоритмы за константное время, чтобы защититься от атак по времени (атакующая сторона пытается определить ваш алгоритм шифрования на основе времени, которое требуется для выполнения криптографических алгоритмов).

Модуль hashlib может использоваться при генерации хэшированных паролей для безопасного хранилища или контрольных сумм с целью подтверждения сохранности данных во время передачи. Функция Password-Based Key Derivation Function 2 (PBKDF2), рекомендованная в NIST Special Publication 800-132 (http://bit.ly/nist-recommendation), в данный момент считается лучшим способом хэширования пароля. Рассмотрим пример использования этой функции вместе с salt[1]. При генерации хэшированного пароля используется 10 000 итераций алгоритма Secure Hash Algorithm для 256-битного хэша (SHA-256) (доступные алгоритмы хэширования и переменное количество итераций позволяют программисту сбалансировать устойчивость и желаемую скорость ответа):

```
import os
import hashlib
def hash_password(password, salt_len=16, iterations=10000, encoding='utf-8'):
    salt = os.urandom(salt_len)
    hashed_password = hashlib.pbkdf2_hmac(
        hash_name='sha256',
        password=bytes(password, encoding),
        salt=salt,
        iterations=iterations
    )
    return salt, iterations, hashed_password
```

Библиотека secrets (https://docs.python.org/3.6/library/secrets.html) была предложена в PEP 506 (https://www.python.org/dev/peps/pep-0506/), она доступна с версии Python 3.6. Предоставляет функции генерации токенов для безопасности, которые подходят приложениям, а также функции восстановления пароля и создания URL, которые сложно угадать. Ее документация содержит примеры и рекомендации по управлению безопасностью на базовом уровне.

pyOpenSSL

Когда вышла библиотека Cryptography, pyOpenSSL (https://pyopenssl.readthedocs.io/en/stable/) обновила свои привязки так, чтобы использовать основанные на CFFI привязки Cryptography для OpenSSL и попасть под крыло PyCA.

[1] salt — это случайная строка, которая еще больше обфусцирует хэш; если бы все использовали один и тот же алгоритм, nefarious actor смог бы сгенерировать таблицу, содержащую распространенные пароли и их хэши, и применить их для «декодирования» украденных файлов с паролями. Чтобы это предотвратить, к паролю прикрепляется случайная строка (salt), так что ее придется хранить для использования в будущем.

pyOpenSSL намеренно не является частью стандартной библиотеки Python, чтобы можно было выпускать обновления безопасности с желаемой скоростью[1] — ее строят для новых версий OpenSSL, а не для версий OpenSSL, поставляющихся с вашей операционной системой (если только вы не строите ее сами для новой версии). Как правило, если вы строите сервер, то используете pyOpenSSL (обратитесь к документации для SSL от Twisted по адресу http://twistedmatrix.com/documents/12.0.0/core/howto/ssl.html — там вы найдете пример применения pyOpenSSL).

Установите ее с помощью `pip`:

```
$ pip install pyOpenSSL
```

И импортируйте под именем OpenSSL. В этом примере показываются несколько доступных функций:

```
>>> import OpenSSL
>>>
>>> OpenSSL.crypto.get_elliptic_curve('Oakley-EC2N-3')
<Curve 'Oakley-EC2N-3'>
>>>
>>> OpenSSL.SSL.Context(OpenSSL.SSL.TLSv1_2_METHOD)
<OpenSSL.SSL.Context object at 0x10d778ef0>
```

Команда разработчиков pyOpenSSL поддерживает код примера (https://github.com/pyca/pyopenssl/tree/master/examples), который включает в себя генерацию сертификатов, способ начать использовать SSL вместо уже соединенного сокета, а также безопасный сервер XMLRPC.

PyNaCl и libnacl

Идея, лежащая в основе libsodium (http://bit.ly/introducing-sodium) (библиотеки-бэкенда, написанной на C, для PyNaCl и libnacl), заключается в том, чтобы намеренно не давать пользователям выбор — лишь лучшие варианты из доступных в их ситуации. Она *не* поддерживает полностью протокол TLS; если вы хотите использовать этот протокол по максимуму, выбирайте pyOpenSSL. Если вам требуется лишь устанавливать зашифрованные соединения с другими компьютерами, которыми вы управляете, или с выбранными вами протоколами и вы не хотите работать с OpenSSL, эта библиотека отлично подойдет[2].

---

[1] Любой человек может подписаться на рассылку cryptography-dev от PyCA, чтобы быть в курсе последних разработок и других новостей. Существует также рассылка новостей OpenSSL.

[2] Если вы хотите полностью управлять вашим кодом, отвечающим за шифрование, и вам неважно, что он работает чуть медленнее, и не нужны самые свежие алгоритмы, попробуйте библиотеку TweetNaCl (https://tweetnacl.cr.yp.to/), которая состоит из одного файла и помещается в сотню твитов. Поскольку релиз PyNaCl поставляется вместе с libsodium, вы, скорее всего, можете использовать TweetNaCl и в то же время запускать практически все (правда, мы сами не пробовали так делать).

PyNaCl произносится py-salt («пай-солт»), а libnacl — lib-salt («либ-солт»). Они обе созданы на основе библиотеки NaCl (salt) (https://nacl.cr.yp.to/).

Мы рекомендуем использовать PyNaCl (https://pypi.python.org/pypi/PyNaCl) вместо libnacl (https://libnacl.readthedocs.io/), поскольку за ней присматривает PyCA и вам не нужно отдельно устанавливать libsodium. Библиотеки, по сути, одинаковы: PyNaCl используют привязки CFFI для библиотек, написанных на C, а libnacl — ctypes (поэтому выбор библиотеки не имеет особого значения). Установите PyNaCl с помощью `pip`:

```
$ pip install PyNaCl
```

В документации к PyNaCl по адресу https://pynacl.readthedocs.io/en/latest/ есть и примеры.

Cryptography

Cryptography (https://cryptography.io/en/latest/) предоставляет рецепты и примитивы для шифрования. Поддерживает Python версий 2.6–2.7 и 3.3+, а также PyPy. PyCA рекомендует в большинстве случаев пользоваться высокоуровневым интерфейсом pyOpenSSL.

Cryptography состоит из двух уровней: рецептов и опасных материалов (hazardous materials, hazmat). Уровень рецептов предоставляет простой API для выполнения качественного симметричного шифрования, а уровень hazmat — низкоуровневые криптографические примитивы. Установите ее с помощью `pip`:

```
$ pip install cryptography
```

В этом примере используется высокоуровневый рецепт симметричного шифрования — единственная высокоуровневая функция этой библиотеки:

```
from cryptography.fernet import Fernet
key = Fernet.generate_key()
cipher_suite = Fernet(key)
cipher_text = cipher_suite.encrypt(b"A really secret message.")
plain_text = cipher_suite.decrypt(cipher_text)
```

PyCrypto

PyCrypto (https://www.dlitz.net/software/pycrypto/) предоставляет безопасные хэш-функции, а также разнообразные алгоритмы шифрования. Поддерживает версии Python 2.1+ и Python 3+. Поскольку код, написанный на C, является пользовательским, PyCA осторожно работайте с библиотекой (но она использовалась де-факто для решения задач, связанных с шифрованием, многие годы, поэтому вы можете встретить ее в более старом коде). Установите ее с помощью `pip`:

```
$ pip install pycrypto
```

Использовать ее можно так:

```
from Crypto.Cipher import AES
# Шифрование
encryption_suite = AES.new('This is a key123', AES.MODE_CBC,
                   'This is an IV456')
cipher_text = encryption_suite.encrypt("A really secret message.")
# Дешифрование
decryption_suite = AES.new('This is a key123', AES.MODE_CBC,
                   'This is an IV456')
plain_text = decryption_suite.decrypt(cipher_text)
```

bcrypt

Если вы хотите применять алгоритм bcrypt (https://en.wikipedia.org/wiki/Bcrypt) для ваших паролей, задействуйте эту библиотеку. Тем, кто раньше пользовался py-bcrypt, должно быть нетрудно перейти на нее, поскольку библиотеки совместимы. Установите ее с помощью `pip`:

```
pip install bcrypt
```

Она имеет всего две функции: `bcrypt.hashpw()` и `bcrypt.gensalt()`. Последняя позволяет выбирать количество итераций — чем больше итераций, тем медленнее работает алгоритм (по умолчанию задается их разумное количество). Рассмотрим пример:

```
>>> import bcrypt
>>>>
>>> password = bytes('password', 'utf-8')
>>> hashed_pw = bcrypt.hashpw(password, bcrypt.gensalt(14))
>>> hashed_pw
b'$2b$14$qAmVOCfEmHeC8Wd5BoF1W.7ny9M7CSZpOR5WPvdKFXDbkkX8rGJ.e'
```

Сохраняем хэшированный пароль:

```
>>> import binascii
>>> hexed_hashed_pw = binascii.hexlify(hashed_pw)
>>> store_password(user_id=42, password=hexed_hashed_pw)
```

Когда приходит время проверять пароль, используйте хэшированный пароль в качестве второго аргумента функции `bcrypt.hashpw()` следующим образом:

```
>>> hexed_hashed_pw = retieve_password(user_id=42)
>>> hashed_pw = binascii.unhexlify(hexed_hashed_pw)
>>>
>>> bcrypt.hashpw(password, hashed_pw)
b'$2b$14$qAmVOCfEmHeC8Wd5BoF1W.7ny9M7CSZpOR5WPvdKFXDbkkX8rGJ.e'
>>>
>>> bcrypt.hashpw(password, hashed_pw) == hashed_pw
True
```

10 Манипуляции с данными

В этой главе приводятся популярные библиотеки Python, предназначенные для работы с данными, которые могут быть численными, текстовыми, а также изображениями и аудио. Практически все эти библиотеки используются для решения уникальных задач, поэтому в этой главе мы лишь опишем их, не сравнивая друг с другом. Вы можете установить библиотеки из PyPI с помощью команды `pip`, если не указано иное:

```
$ pip install library
```

В табл. 10.1 кратко описываются библиотеки.

Таблица 10.1. Инструменты для работы с данными

Библиотека Python	Лицензия	Причины использовать
IPython	Лицензия Apache 2.0	Предоставляет улучшенный интерпретатор Python, имеющий историю ввода, интегрированный отладчик, а также возможность строить графики в терминале (с помощью версии с Qt)
NumPy	Лицензия BSD 3-clause	Предоставляет многомерные массивы и инструменты линейной алгебры, оптимизированные для скорости
SciPy	Лицензия BSD	Предоставляет функции и вспомогательные программы, связанные с инженерией и наукой (от линейной алгебры до обработки сигналов), интеграцией, поиском корня, статистическим распределением и другими темами
Matplotlib	Лицензия BSD	Позволяет строить научные графики

Продолжение ⇗

Таблица 10.1 *(продолжение)*

Библиотека Python	Лицензия	Причины использовать
Pandas	Лицензия BSD	Предоставляет ряды и объекты DataFrame, которые можно сохранять, объединять, группировать, выполнять агрегацию, индексировать окна и создавать их подмножества — очень похоже на R Data Frame или содержимое запроса SQL
Scikit-Learn	Лицензия BSD 3-clause	Предоставляет алгоритмы машинной обработки, включающие понижение размерности, регрессию, работу с кластерами, выбор модели, ввод недостающих данных и предварительную обработку
Rpy2	Лицензия GPLv2	Предоставляет интерфейс к R, который позволяет выполнять функции этого языка изнутри Python, а также передавать данные между двумя средами
SymPy	Лицензия BSD	Предоставляет символьную математику, включающую разложение в ряд, пределы и анализ. Стремится выглядеть как полноценная вычислительная система
nltk	Лицензия Apache	Предоставляет полноценный натуральный тулкит, имеющий модели и данные для обучения на многих языках
pillow / PIL	Стандартная лицензия PIL (наподобие лицензии MIT)	Предоставляет огромное количество форматов файлов, а также возможности по фильтрации изображений и прочей обработке
cv2	Лицензия Apache 2.0	Предоставляет программы для машинного распознавания образов, подходящие для выполнения анализа видеороликов в реальном времени, включая заранее обученные алгоритмы определения людей и лиц
Scikit-Image	Лицензия BSD	Предоставляет подпрограммы для обработки изображений (фильтрации, регулирования, цветоделения, определения краев, пятен, углов, сегментации и пр.)

Практически все эти библиотеки зависят от библиотек, написанных на C, в частности от SciPy или одной из ее зависимостей NumPy. Это означает, что у вас могут возникнуть проблемы с их установкой, если вы работаете в ОС Windows. Если вы используете Python в основном для анализа научных данных и не знакомы с компилированием кода на C и FORTRAN в ОС Windows, рекомендуем выбрать Anaconda или один из других вариантов, представленных в разделе «Коммерческие дистрибутивы Python» главы 2. В противном случае всегда сначала пробуйте выполнять команду `pip install`, а если она даст сбой, обращайтесь к руководству по установке SciPy (https://www.scipy.org/install.html).

Научные приложения

Python часто используется для создания высокопроизводительных научных приложений. Широко применяется в академических и научных проектах, поскольку код на нем легко писать и язык имеет высокую производительность. В Python для научных вычислений зачастую используются внешние библиотеки, обычно написанные на более быстрых языках (вроде C или FORTRAN для работы с матрицами). Основные используемые библиотеки — части «стека SciPy»: NumPy, SciPy, SymPy, Pandas, Matplotlib и IPython. Подробное знакомство с ними выходит за рамки темы этой книги. Однако вы можете найти полное введение для экосистемы научного Python в Python Scientific Lecture Notes (http://scipy-lectures.github.com/).

IPython

IPython (http://ipython.org/) — это улучшенная версия интерпретатора Python, имеющая цветной интерфейс, более подробные сообщения об ошибках и *режим встраивания*, который позволяет отображать графики в терминале (в версии на основе Qt). Он является ядром по умолчанию для Jupyter (рассматриваются в разделе «Jupyter Notebooks» главы 7), а также интерпретатором по умолчанию для Spyder IDE (рассматривается в подразделе «Spyder» раздела «IDE» главы 3). IPython поставляется вместе с Anaconda, описанной в разделе «Коммерческие дистрибутивы Python» главы 2.

NumPy

NumPy (http://numpy.scipy.org/) является частью проекта SciPy, но она выпущена как отдельная библиотека, поэтому те, кому нужна лишь базовая функциональность, могут использовать ее, не устанавливая остальную часть SciPy.

NumPy с умом обходит проблему запуска более медленных алгоритмов в Python путем использования многомерных массивов и функций, которые работают с массивами. Любой алгоритм можно представить как функцию для массивов, что позволяет запускать алгоритмы быстро. Бэкендом выступает библиотека Automatically Tuned Linear Algebra Software (ATLAS)[1] (http://math-atlas.sourceforge.net/), а также другие низкоуровневые библиотеки, написанные на C и FORTRAN. NumPy совместима с версиями Python 2.6+ и 3.2+.

Рассмотрим пример умножения матриц с помощью `array.dot()`, а также «трансляции», представляющей собой поэлементное умножение, где строка или столбец повторяются для отсутствующих измерений:

[1] ATLAS — это развивающийся программный проект, который предоставляет протестированные производительные библиотеки для работы с линейной алгеброй. Он предоставляет интерфейсы на языках C и FORTRAN 77 для программ из хорошо известных Basic Linear Algebra Subset (BLAS) и Linear Algebra PACKage (LAPACK).

```
>>> import numpy as np
>>>
>>> x = np.array([[1,2,3],[4,5,6]])
>>> x array([[1, 2, 3],
       [4, 5, 6]])
>>>
>>> x.dot([2,2,1])
array([ 9, 24])
>>>
>>> x  * [[1],[0]]
array([[1, 2, 3],
       [0, 0, 0]])
```

SciPy

SciPy (http://scipy.org/) использует NumPy для выполнения математических функций. SciPy задействует массивы NumPy в качестве базовой структуры данных. Она поставляется с модулями для решения распространенных задач научного программирования, включая линейную алгебру, анализ, особые функции и константы, а также обработку сигналов.

Рассмотрим пример, в котором используются физические константы SciPy:

```
>>> import scipy.constants
>>> fahrenheit = 212
>>> scipy.constants.F2C(fahrenheit)
100.0
>>> scipy.constants.physical_constants['electron mass']
(9.10938356e-31, 'kg', 1.1e-38)
```

Matplotlib

Matplotlib (http://matplotlib.sourceforge.net/) — это гибкая библиотека для сборки интерактивных 2D и 3D графиков, которые также могут быть сохранены как собранные вручную численные показатели. API во многом отражает API MATLAB (http://www.mathworks.com/products/matlab/), что упрощает переход пользователей MATLAB на Python. В галерее Matplotlib по адресу http://matplotlib.sourceforge.net/gallery.html содержится множество примеров, а также исходный код к ним, что позволяет воссоздать их самостоятельно.

Тем, кто работает со статистикой, можно взглянуть на Seaborn (https://stanford.edu/~mwaskom/software/seaborn), более новую библиотеку для работы с графикой, предназначенную для визуализации статистики. О ней рассказывается в этой статье, посвященной тому, как освоить науку о данных (http://bit.ly/data-science-python-guide).

Для того чтобы строить графики, задействуйте Bokeh (http://bokeh.pydata.org/), использующую собственные библиотеки для визуализации, или Plotly (https://plot.ly/),

основанную на библиотеке D3.js (https://d3js.org/), написанной на JavaScript, однако бесплатная версия Plotly может потребовать, чтобы вы хранили свои графики на их сервере.

Pandas

Pandas (http://pandas.pydata.org/) (имя основано на фразе Panel Data — «панель с данными») — это библиотека, предназначенная для манипуляций с данными. Основана на NumPy, которая предоставляет множество полезных функций для получения доступа, индексирования, объединения и группирования данных. Основная структура данных (`DataFrame`) похожа на структуру, которую можно найти в среде статистического ПО R (то есть гетерогенные таблицы данных — имеющие в одних столбцах строки, а в других числа — с возможностью индексирования имени, операций с временными рядами, а также автоматического выстраивания данных). С ней также можно работать как с таблицей SQL или Excel Pivot Table, используя методы вроде `groupby()` или функции вроде `pandas.rolling_mean()`.

Scikit-Learn

Scikit-Learn (https://pypi.python.org/pypi/scikit-learn) — это библиотека, посвященная машинному обучению, которая предоставляет способы понижения размерности, заполнение отсутствующих данных, регрессию и модели классификации, модели деревьев, кластеры, автоматическую подстройку параметров моделей, построение графиков (с помощью Matplotlib) и многое другое. Она хорошо задокументирована и поставляется с огромным количеством примеров (http://scikit-learn.org/stable/auto_examples/index.html). Scikit-Learn работает с массивами NumPy, но обычно может взаимодействовать с порциями данных от Panda без особых проблем.

Rpy2

Rpy2 (https://pypi.python.org/pypi/rpy2) — это привязка к Python для статистического пакета R, позволяющая выполнять функции R из кода Python и передавать данные между двумя средами. Rpy2 — это объектно-ориентированная реализация привязки Rpy (http://rpy2.bitbucket.org/).

decimal, fractions и numbers

В Python определен фреймворк абстрактных базовых классов, предназначенный для разработки численных типов — от типа `Number`, который является основным численным типом, до типов `Integral`, `Rational`, `Real` и `Complex`. Разработчики могут создавать подклассы для этих классов при разработке других численных типов

в соответствии с инструкциями, приведенными в библиотеке numbers[1] (https://docs.python.org/3.5/library/numbers.html). Существует также класс `decimal.Decimal`, который отслеживает численную точность (предназначен для бухгалтерского дела и других задач, где требуется точность). Иерархия типов работает в соответствии с ожиданиями:

```
>>> import decimal
>>> import fractions
>>> from numbers import Complex, Real, Rational, Integral
>>>
>>> d = decimal.Decimal(1.11, decimal.Context(prec=5))  # precision
>>>
>>> for x in (3, fractions.Fraction(2,3), 2.7, complex(1,2), d):
...     print('{:>10}'.format(str(x)[:8]),
...           [isinstance(x, y) for y in (Complex, Real, Rational, Integral)])
...
         3 [True, True, True, True]
       2/3 [True, True, True, False]
       2.7 [True, True, False, False]
    (1+2j) [True, False, False, False]
  1.110000 [False, False, False, False]
```

Экспоненциальные, тригонометрические и другие распространенные функции находятся в библиотеке math, а соответствующие функции для комплексных чисел — в библиотеке cmath. Библиотека random предоставляет псевдослучайные числа, используя в качестве основного генератора Mersenne Twister (https://en.wikipedia.org/wiki/Mersenne_Twister). На момент выхода версии Python 3.4 модуль statistics стандартной библиотеки предоставляет возможность определить среднее значение и медиану, а также квадратичное отклонение и дисперсию для выборки и совокупности.

SymPy

SymPy (https://pypi.python.org/pypi/sympy) — это библиотека, которую следует использовать при работе с символьной математикой в Python. Она полностью написана на Python и имеет опциональные расширения для ускорения работы, а также построения графиков и интерактивных сессий.

Символьные функции SymPy работают с объектами SymPy, такими как символы, функции и выражения, для создания других символьных выражений, например так:

[1] Одним из популярных инструментов, использующим числа Python, является SageMath (http://www.sagemath.org/) — всеобъемлющий инструмент, который определяет классы для представления полей, колец, алгебр и доменов, а также предоставляет символьные инструменты, унаследованные из SymPy, и численные инструменты, унаследованные из NumPy, SciPy и многих других библиотек, написанных как на Python, так и на других языках.

```
>>> import sympy as sym
>>>
>>> x = sym.Symbol('x')
>>> f = sym.exp(-x**2/2) / sym.sqrt(2 * sym.pi)
>>> f sqrt(2)*exp(-x**2/2)/(2*sqrt(pi))
```

Их можно интегрировать как символьно, так и численно:

```
>>> sym.integrate(f, x)
erf(sqrt(2)*x/2)/2
>>>
>>> sym.N(sym.integrate(f, (x, -1, 1)))
0.682689492137086
```

Библиотека также может брать производную, раскладывать выражения в ряды, ограничивать доступные символы действительными, коммутационными или соответствующими десятку других категорий, находить ближайшее рациональное число (с заданной точностью) для числа с плавающей точкой и многое другое.

Манипуляции с текстом и его анализ

Инструменты для работы со строками в Python — одна из причин, почему многие начинают использовать язык. Мы кратко рассмотрим основные инструменты из стандартной библиотеки Python, а затем перейдем к библиотеке, которую применяют практически все члены сообщества для анализа текста: Natural Language ToolKit (nltk) (https://pypi.python.org/pypi/nltk).

Инструменты для работы со строками стандартной библиотеки Python

Если в языке имеются символы, которые ведут себя особенным образом, когда записаны в нижнем регистре, работать с ними поможет метод `str.casefold()`:

```
>>> 'Grünwalder Straße'.upper()
'GRÜNWALDER STRASSE'
>>> 'Grünwalder Straße'.lower()
'grünwalder straße'
>>> 'Grünwalder Straße'.casefold()
'grünwalder strasse'
```

Библиотека Python для работы с регулярными выражениями всеобъемлющая и мощная — мы видели ее в действии в пункте «Регулярные выражения (читаемость имеет значение)» на с. 177, поэтому мы не будем рассматривать ее подробно. Отметим лишь, что документация, которую можно получить с помощью вызова `help(re)`, достаточно информативна, так что вам не придется открывать браузер во время написания кода.

Модуль difflib из стандартной библиотеки позволяет определить разницу между строками и имеет функцию `get_close_matches()`, которая может помочь при опечатках, когда существует известный набор правильных ответов (например, для сообщений об ошибке на сайте, посвященном путешествиям):

```
>>> import difflib
>>> capitals = ('Montgomery', 'Juneau', 'Phoenix', 'Little Rock')
>>> difflib.get_close_matches('Fenix', capitals)
['Phoenix']
```

nltk

Natural Language ToolKit (nltk) — это инструмент для анализа текста. Изначально выпущен Стивеном Бердом (Steven Bird) и Эдвардом Лопером (Edward Loper), чтобы помочь студентам усвоить курс Берда о Natural Language Processing (NLP), преподававшийся в University of Pennsylvania в 2001 году. Со временем вырос до размеров большой библиотеки, покрывающей множество языков и содержащей алгоритмы, связанные с последними исследованиями в области. Доступен под лицензией Apache 2.0, его загружают из PyPI более 100 000 раз в месяц. Его создатели выпустили книгу *Natural Language Processing with Python* (издательство O'Reilly): информация изложена в виде курса, который познакомит вас с Python и NLP.

Вы можете установить nltk из командной строки с помощью pip[1]. Он полагается на библиотеку NumPy, поэтому сначала установите ее:

```
$ pip install numpy
$ pip install nltk
```

Если вы используете Windows и не можете заставить работать NumPy, установленный с помощью `pip`, можете попробовать выполнить инструкции, приведенные по адресу http://bit.ly/numpy-install-win, на ресурсе Stack Overflow.

Размер и область видимости библиотеки могут отпугнуть некоторых пользователей, поэтому рассмотрим небольшой пример, иллюстрирующий, насколько просто работать с этим инструментом. Для начала нам понадобится получить набор данных (http://www.nltk.org/data.html) из отдельно загружаемого набора корпусов (http://www.nltk.org/nltk_data/), включая инструменты для тегирования для нескольких языков и набора данных, на которых будут тестироваться алгоритмы. Они имеют лицензию, отличающуюся от лицензии nltk, поэтому убедитесь, что вы проверили лицензию выбранного набора данных. Если знаете название корпуса

[1] В операционной системе Windows на момент написания книги nltk доступен только для Python 2.7. Попробуйте запустить его на Python 3; метки «Python 2.7» могли устареть.

текста, который нужно загрузить (в нашем случае это Punkt tokenizer[1], который мы можем использовать для разбиения текстовых файлов на предложения или отдельные слова), можете сделать это с помощью командной строки:

```
$ python3 -m nltk.downloader punkt --dir=/usr/local/share/nltk_data
```

Или можете загрузить его в рамках интерактивной сессии — stopwords содержит список слов, из-за которых значительно увеличивается общее количество слов текста вроде the, in или and во многих языках:

```
>>> import nltk
>>> nltk.download('stopwords', download_dir='/usr/local/share/nltk_data')
[nltk_data] Downloading package stopwords to /usr/local/share/nltk_data...
[nltk_data]   Unzipping corpora/stopwords.zip.
True
```

Если вы не знаете название необходимого вам корпуса, можете запустить интерактивный загрузчик из интерпретатора Python, вызвав метод `nltk.download()` без передачи первого аргумента:

```
>>> import nltk
>>> nltk.download(download_dir='/usr/local/share/nltk_data')
```

Далее можно загрузить самую свежую версию набора данных и запустить ее в обработку. В этом фрагменте кода мы загружаем сохраненную копию «Дзена Питона»:

```
>>> import nltk
>>> from nltk.corpus import stopwords
>>> import string
>>>
>>> stopwords.ensure_loaded()  ❶
>>> text = open('zen.txt').read()
>>> tokens = [
...     t.casefold() for t in nltk.tokenize.word_tokenize(text)  ❷
...     if t not in string.punctuation
... ]
>>>
>>> counter = {}
>>> for bigram in nltk.bigrams(tokens):  ❸
...     counter[bigram] = 1 if bigram not in counter
                           else counter[bigram] + 1
...
>>> def print_counts(counter):  # We'll reuse this
```

[1] Алгоритм Punkt tokenizer создан Тибором Киссом (Tibor Kiss) и Яном Странком (Jan Strunk) в 2006 году (http://bit.ly/kiss-strunk-paper). Представляет собой не зависящий от языка способ определять границы предложения, например конструкция Mrs. Smith and Johann S. Bach listened to Vivaldi будет корректно определена как одно предложение. Алгоритм необходимо обучить на большом наборе данных, но алгоритм для английского языка, используемый по умолчанию, уже обучен.

```
...     for ngram, count in sorted(
...             counter.items(), key=lambda kv: kv[1],  ❹
...             reverse=True):
...         if count > 1:
...             print ('{:>25}: {}'.format(str(ngram),  ❺
...             '*' * count))
...
>>> print_counts(counter)
       ('better', 'than'): ********  ❻
         ('is', 'better'): *******
        ('explain', 'it'): **
            ('one', '--'): **
        ('to', 'explain'): **
            ('if', 'the'): **
('the', 'implementation'): **
 ('implementation', 'is'): **
>>>
>>> kept_tokens = [t for t in tokens if t not in stopwords.words()]  ❼
>>>
>>> from collections import Counter  ❽
>>> c = Counter(kept_tokens)
>>> c.most_common(5)
[('better', 8), ('one', 3), ('--', 3), ('although', 3), ('never', 3)]
```

❶ Корпуса загружаются медленно, поэтому нам нужно сделать это для того, чтобы действительно загрузить корпус stopwords.

❷ Токенизатор требует наличия обученной модели — Punkt tokenizer (используемый по умолчанию) поставляется с моделью, обученной для английского языка (также выбран по умолчанию).

❸ Биграмма — это пара соседних слов. Мы проходим по биграммам и считаем, сколько раз они встречаются.

❹ Ключом для функции `sorted()` является количество элементов, они отсортированы в обратном порядке.

❺ Конструкция `'{:>25}'` выравнивает справа строку с общей длиной, равной 25 символам.

❻ Наиболее часто встречающейся биграммой «Дзена Питона» является фраза better than («лучше, чем»).

❼ В этот раз для того, чтобы избежать большого количества слов the и is, мы удалим stopwords.

❽ В версиях Python 3.1 и выше для подсчета можно использовать метод `collections.Counter`.

В этой библиотеке еще много интересного — выделите выходные и исследуйте ее!

SyntaxNet

Библиотека SyntaxNet от Google, созданная на основе TensorFlow, предоставляет обученный анализатор для английского языка (по имени Parsey McParseface) и фреймворк для сборки других моделей, даже для других языков, если у вас будут под рукой соответствующие данные. В настоящий момент библиотека доступна только для Python 2.7; подробные инстуркции по ее загрузке и использованию вы можете найти на странице https://github.com/tensorflow/models/tree/master/syntaxnet.

Работа с изображениями

Тремя наиболее популярными библиотеками для обработки изображений и выполнения действий с ними в Python являются Pillow (дружественная параллельная версия библиотеки Python Imaging Library (PIL), которая подходит для преобразования форматов и простой обработки изображений), cv2 (привязка к Python для библиотеки Open-Source Computer Vision (OpenCV), которую можно использовать для определения лиц в реальном времени, а также для реализации других продвинутых алгоритмов) и более новая Scikit-Image (предоставляет возможности по простой обработке изображений, а также примитивы вроде пятен и фигур и функциональность для обнаружения границ). В следующих разделах приведена более подробная информация о каждой из них.

Pillow

Python Imaging Library (PIL) (http://www.pythonware.com/products/pil/) — одна из основных библиотек для выполнения различных действий с изображениями в Python. Последняя ее версия выпущена в 2009 году, она не была портирована на Python 3. К счастью, активно разрабатывается параллельная версия, которая называется Pillow (http://python-pillow.github.io/) (ее проще устанавливать, она работает во всех операционных системах и поддерживает Python 3).

Перед установкой Pillow вам нужно установить ее зависимости. Более подробные инструкции для своей платформы вы можете найти по адресу https://pillow.readthedocs.org/en/3.0.0/installation.html (после этого все выглядит довольно понятно):

```
$ pip install Pillow
```

Рассмотрим пример использования Pillow (для команды `import from` применяется имя PIL, а не Pillow):

```
from PIL import Image, ImageFilter
# Считываем изображение
im = Image.open( 'image.jpg' )
```

```
# Показываем изображение
im.show()
# Применяем фильтр к изображению
im_sharp = im.filter( ImageFilter.SHARPEN )
# Сохраняем отфильтрованное изображение в новый файл
im_sharp.save( 'image_sharpened.jpg', 'JPEG' )
# Разбиваем изображение на соответствующие bands (то есть на красный, зеленый
# и синий для RGB)
r,g,b = im_sharp.split()
# Просматриваем данные EXIF, встроенные в изображение
exif_data = im._getexif()
exif_data
```

Другие примеры применения библиотеки Pillow смотрите в руководстве к ней по адресу http://bit.ly/opencv-python-tutorial.

cv2

Библиотека OpenSource Computer Vision (OpenCV) (http://docs.opencv.org/3.1.0/index.html) предлагает более широкие возможности для работы с изображениями и их обработки, нежели PIL. Написана на C и C++ и концентрируется на распознавании образов машиной в реальном времени. Например, она содержит первую модель, использованную при распознавании лиц в реальном времени (уже обученная на тысячах лиц; в примере по адресу https://github.com/Itseez/opencv/blob/master/samples/python/facedetect.py показывается ее применение в коде Python), модель распознавания лиц, а также модель распознавания людей среди всего остального. Реализована на нескольких языках и распространена повсеместно.

В Python обработка изображений с помощью OpenCV реализована с использованием библиотек cv2 и NumPy. Третья версия OpenCV имеет связки для версий Python 3.4 и выше, но библиотека cv2 все еще связана с OpenCV2, которая не имеет привязки к этим версиям Python. Инструкции по установке, размещенные по адресу http://tinyurl.com/opencv3-py-tutorial, содержат подробную информацию для ОС Windows и Fedora, используется версия Python 2.7. Если вы работаете с OS X, то вы сами по себе[1]. Наконец, существует вариант установки для ОС Ubuntu с использованием Python 3 (http://tinyurl.com/opencv3-py3-ubuntu). Если процесс установки станет сложным, вы можете загрузить Anaconda; они имеют бинарные файлы cv2 для всех платформ (можете прочесть статью Up & Running: OpenCV3, Python 3,

[1] Эти шаги сработали для нас: для начала используйте команду brew install opencv или brew install opencv3 --with-python3. Далее следуйте инструкциям (вроде связывания с NumPy). Наконец, добавьте каталог, содержащий общий файл объекта OpenCV (например, /usr/local/Cellar/opencv3/3.1.0_3/lib/python3.4/site-packages/) к вашему пути; или введите команду add2virtualenvironment (http://virtualenvwrapper.readthedocs.io/en/latest/command_ref.html#add2virtualenv), установленную вместе с библиотекой virtualenvwrapper, если собираетесь использовать библиотеку только в виртуальной среде.

& Anaconda по адресу http://tinyurl.com/opencv3-py3-anaconda, чтобы узнать, как применять cv2 и Python 3 в Anaconda).

Рассмотрим пример использования cv2:

```
from cv2 import *
import numpy as np
# Считываем изображение
img = cv2.imread('testimg.jpg')
# Показываем изображение
cv2.imshow('image',img)
cv2.waitKey(0)
cv2.destroyAllWindows()
# Применяем к изображению фильтр Grayscale
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
# Сохраняем отфильтрованное изображение в новый файл
cv2.imwrite('graytest.jpg',gray)
```

В сборнике руководств к OpenCV по адресу http://opencv-python-tutroals.readthedocs.org/en/latest/py_tutorials/py_tutorials.html представлено еще больше примеров для Python.

Scikit-Image

Популярность более новой библиотеки Scikit-Image (http://scikit-image.org/) растет отчасти благодаря тому, что большая часть ее исходного кода написана на Python, также она имеет отличную документацию. У нее нет полнофункциональных алгоритмов, как cv2, которую вы все еще можете использовать для алгоритмов, работающих с видео в реальном времени, но она полезна для ученых (например, они используют функции вроде определения пятен). Кроме того, библиотека включает инструменты для стандартной обработки изображений вроде фильтрации и настройки контрастности. Например, Scikit-image использовалась для создания изображений малых лун Плутона (https://blogs.nasa.gov/pluto/2015/10/05/plutos-small-moons-nix-and-hydra/). На основной странице Scikit-Image можно найти дополнительные примеры (http://scikit-image.org/docs/dev/auto_examples/).

11 Хранение данных

Мы уже упоминали сжатие ZIP и сериализацию в разделе «Сериализация данных» главы 9, поэтому в этой главе нам осталось рассмотреть только базы данных.

Глава посвящена библиотекам Python, которые взаимодействуют с *реляционными базами данных*. Как правило, когда речь идет о базах данных, мы думаем именно о такой их разновидности — они содержат данные, сохраненные в таблицах, мы получаем к ним доступ с помощью SQL[1].

Структурированные файлы

Мы уже упоминали инструменты для JSON, XML и ZIP-файлов в главе 9, а также сериализацию и XDR, когда говорили о сериализации. Для анализа YAML мы рекомендуем PyYAML (http://pyyaml.org/wiki/PyYAML) (вы можете получить его с помощью команды `pip install pyyaml`). В стандартной библиотеке Python также имеются инструменты для работы с файлами CSV, `*.netrc`, используемыми некоторыми клиентами FTP, файлами `*.plist`, применяемыми в OS X, а также файлами, содержащими код на диалекте формата INI из Windows с помощью модуля configparser[2].

---

[1] Идея реляционных баз данных предложена в 1970 году Эдгаром Ф. Коддом (Edgar F. Codd), работником компании IBM. Он написал статью «A Relational Model of Data for Large Share Data Banks» (http://bit.ly/relational-model-data), которой не интересовались до 1977-го, пока Ларри Эллисон (Larry Ellison) не создал компанию (она в итоге стала компанией Oracle), взявшую за основу эту технологию. Другие идеи-конкуренты вроде хранилища, работающие по принципу «ключ-значение», и иерархические модели баз данных игнорировались после успеха реляционных баз данных. Лишь недавно движение not only SQL (NoSQL) возродило идею использовать нереляционные хранилища для кластерных вычислений.

[2] В Python2 это ConfigParser; обратитесь к документации для configparser (https://docs.python.org/3/library/configparser.html#supported-ini-file-structure), чтобы увидеть точный диалект, с которым работает анализатор.

Существует также устойчивое хранилище, работающее по принципу «ключ-значение», доступное благодаря модулю `shelve` из стандартной библиотеки Python. Его бэкенд является наилучшим доступным вариантом менеджера базы данных (dbm — база данных, работающая по принципу «ключ-значение») для вашего компьютера[1]:

```
>>> import shelve
>>>
>>> with shelve.open('my_shelf') as s:
...     s['d'] = {'key': 'value'}
...
>>> s = shelve.open('my_shelf', 'r')
>>> s['d']
{'key': 'value'}
```

Вы можете узнать, какой бэкенд используете, таким способом:

```
>>> import dbm
>>> dbm.whichdb('my_shelf')
'dbm.gnu'
```

Вы также можете загрузить реализацию GNU для dbm для Windows с сайта http://gnuwin32.sourceforge.net/packages/gdbm.htm или сначала проверить ее наличие в менеджере пакетов (`brew`, `apt`, `yum`), а затем попробовать установить с помощью исходного кода dbm (http://www.gnu.org.ua/software/gdbm/download.html).

Библиотеки для работы с базами данных

API для работы с базами данных в Python (DB-API2) определяет стандартный интерфейс для доступа к базам данных. Он задокументирован в PEP 249 (https://www.python.org/dev/peps/pep-0249/), а также в более подробном введении к DB-API (http://halfcooked.com/presentations/osdc2006/python_databases.html). Практически все драйверы для баз данных в Python отвечают требованиям этого интерфейса, поэтому, если вы хотите обратиться к базе данных с помощью Python, выберите драйвер, позволяющий соединиться с базой данных, которую вы используете, например sqlite3 для базы данных SQLite, psycopg2 для Postgres и MySQL-python для MySQL[2].

---

[1] Библиотека dbm хранит пары ключ-значение в хэш-таблице, находящейся на диске. Точный механизм ее работы зависит от ее бэкенда — gdbm, ndbm или dumb («глупый»). «Глупый» бэкенд реализован в Python и хорошо задокументирован. Про два других вы можете прочитать в руководстве к gdbm (http://www.gnu.org.ua/software/gdbm/manual//gdbm.html). Для ndbm существует верхняя граница для сохраняемых значений. При открытии файла для записи он блокируется, если (только для gdbm) вы не открываете файл базы данных в режиме ru или wu, и даже тогда обновления могут быть невидимы для других соединений.

[2] Несмотря на то что язык структурированных запросов (Structured Query Language, SQL) является стандартом ISO (http://bit.ly/sql-iso-standard), поставщики баз данных сами выбирают, насколько полно его реализовывать, а также могут добавить собственную функциональность. Это означает, что библиотека Python, которая служит драйвером базы данных, должна понимать диалект SQL выбранной вами базы данных.

Код, содержащий большое количество строк SQL, а также жестко закодированные столбцы и таблицы, быстро становится неопрятным, подвержен ошибкам, его сложно отлаживать. Библиотеки, перечисленные в табл. 11.1 (за исключением sqlite3, драйвера для SQLite), предлагают *уровень абстракции базы данных* (database abstraction layer, DAL), который позволяет абстрагироваться от структуры, грамматики и типов данных SQL, чтобы предоставить API.

Поскольку Python — объектно-ориентированный язык, абстракция для базы данных также может реализовать объектно-реляционное отображение (object-relational mapping, ORM), чтобы соотнести объекты Python и базу данных, а также операторы для атрибутов этих классов, которые представляют собой абстрагированную версию SQL в Python.

Все библиотеки, перечисленные в табл. 11.1 (за исключением sqlite3 и Records), предоставляют ORM, их реализации используют один из двух шаблонов[1]: *Active Record* (записи одновременно представляют абстрагированные данные и взаимодействуют с базой данных) и *Data Mapper* (один слой взаимодействует с базой данных, еще один слой представляет данные, а между ними имеется функция соотнесения, которая выполняет логику, необходимую для того, чтобы преобразовывать данные между этими слоями (по сути, выполняет логику представления SQL за пределами базы данных)).

При выполнении запросов шаблоны Active Record и Data Mapper ведут себя примерно одинаково, но, работая с Data Mapper, пользователь должен явно указывать имена таблиц, добавлять первичные ключи и создавать вспомогательные таблицы для поддержки отношений «многие-ко-многим» (например, как в чеке — один идентификатор транзакции будет связан с несколькими покупками); при использовании шаблона Active Record эти действия выполняются за кулисами.

Наиболее популярными библиотеками являются sqlite3, SqlAlchemy и Django ORM. Records находится в собственной категории — это скорее клиент SQL, который предоставляет возможность форматирования выводимой информации; оставшиеся библиотеки можно рассматривать как отдельные легковесные версии Django ORM underneath (поскольку все они используют шаблон ActiveRecord), но с разными реализациями и уникальными API.

---

[1] Эти шаблоны определены в книге Мартина Фаулера (Martin Fowler) Patterns of Enterprise Application Architecture (http://www.martinfowler.com/books/eaa.html). Чтобы подробнее узнать о том, из чего состоят ORM проектов Python, рекомендуем прочесть раздел «SQLAlchemy» книги Architecture of Open Source Applications (http://www.aosabook.org/en/sqlalchemy.html), а также взглянуть на список ссылок, связанных с ORM для Python, предоставленный FullStack Python: https://www.fullstackpython.com/object-relational-mappers-orms.html.

Таблица 11.1. Библиотеки для работы с базами данных

Библиотека	Лицензия	Причины использовать
sqlite3 (драйвер, не ORM)	PSFL	• Находится в стандартной библиотеке. • Подходит для сайтов с низким или умеренным трафиком, для которого требуются более простые типы данных и малое количество запросов, — у него небольшая задержка, поскольку не осуществляется общения по сети. • Подходит для изучения SQL или DB-API для Python, а также для прототипирования приложения, работающего с базами данных
SQLAlchemy	Лицензия MIT	• Предоставляет шаблон Data Mapper, имеющий двухуровневый API, верхний уровень похож на ORM API в других библиотеках, нижний уровень работает с таблицами и непосредственно связан с базой данных. • Явно дает вам возможность контролировать (с помощью API нижнего уровня) структуру и схемы вашей базы данных; это может быть полезно, если, например, ваши базы данных администрируют не веб-разработчики. • Диалекты: SQLite, PostgreSQL, MySQL, Oracle, MS-SQL Server, Firebird и Sybase (также можно зарегистрировать собственный)
Django ORM	Лицензия BSD	• Предоставляет шаблон Active Record, который может неявно сгенерировать инфраструктуру базы данных с помощью определенных пользователем моделей в приложении. • Тесно связан с Django. • Диалекты: SQLite, PostgreSQL, MySQL и Oracle; также вы можете использовать стороннюю библиотеку: SAP SQL Anywhere, IBM DB2, MS-SQL Server, Firebird или ODBC
peewee	Лицензия MIT	• Предоставляет шаблон Active Record, но он работает, поскольку таблицы, которые вы определяете в ORM, вы увидите в базе данных (плюс столбец для индексирования). • Диалекты: SQLite, MySQL и Postgres (а также ваши собственные)
PonyORM	AGPLv3	• Предоставляет шаблон Active Record, а также интуитивный синтаксис, основанный на генераторе. • В сети имеется графический редактор диаграмм «сущность — отношение» (предназначен для рисования модели данных, определяющей таблицы в базе данных, а также их отношения друг с другом), который может быть преобразован в код SQL для создания таблиц. • Диалекты: SQLite, MySQL, Postgres и Oracle (а также ваш собственный)

Продолжение ⇗

Таблица 11.1 *(продолжение)*

Библиотека	Лицензия	Причины использовать
SQLObject	LGPL	• Одним из первых начал использовать шаблон ActiveRecord в Python. • Диалекты: SQLite, MySQL, Postgres, Firebird, Sybase, MAX DB, MS-SQL Server (а также ваш собственный)
Records (интерфейс запросов, не ORM)	Лицензия ISC	• Предоставляет простой способ запрашивать базы данных и генерирует документы отчета: SQL на входе, XLS (или JSON, или YAML, или CSV, или LaTex) на выходе. • Имеет интерфейс командной строки, который может быть использован для интерактивных запросов или генерации отчетов с помощью одной строки. • Использует в качестве бэкенда SQLAlchemy

В следующих разделах предоставляется дополнительная информация о библиотеках из табл. 11.1.

sqlite3

SQLite — это библиотека, написанная на C. Предоставляет базу данных на базе sqlite3 (https://docs.python.org/3/library/sqlite3.html). База данных хранится как один файл, по соглашению он имеет расширение `*.db`. Страница *when to use SQLite* («Когда использовать SQLite») (https://www.sqlite.org/whentouse.html) говорит, что библиотека используется как бэкенд базы данных для сайтов, имеющих сотни тысяч посещений в день. На странице https://www.sqlite.org/lang.html также приведен список команд SQL, которые понимает SQLite. Вы можете проконсультироваться с quick SQL reference от W3Schools (http://www.w3schools.com/sql/sql_quickref.asp), чтобы узнать, как использовать эти команды. Рассмотрим пример:

```
import sqlite3
db = sqlite3.connect('cheese_emporium.db')
db.execute('CREATE TABLE cheese(id INTEGER, name TEXT)')
db.executemany(
    'INSERT INTO cheese VALUES (?, ?)',
    [(1, 'red leicester'),
     (2, 'wensleydale'),
     (3, 'cheddar'),
    ]
)
db.commit()
db.close()
```

Допустимыми типами SQLite являются `NULL`, `INTEGER`, `REAL`, `TEXT` и `BLOB` (`bytes`), также с помощью документации к sqlite3 вы можете зарегистрировать новые типы данных (например, они реализуют тип `datetime.datetime`, который хранится как TEXT).

SQLAlchemy

SQLAlchemy (http://www.sqlalchemy.org/) — очень популярный тулкит для баз данных. Django имеет возможность переключиться с собственного ORM на SQLAchemy. Это бэкенд для мегаруководства для Flask по созданию собственного блога (http://blog.miguelgrinberg.com/post/the-flask-mega-tutorial-part-i-hello-world). Pandas использует его как SQL (http://bit.ly/pandas-sql-query).

SQLAlchemy — это единственная из перечисленных здесь библиотек, следующая шаблону Data Mapper (http://martinfowler.com/eaaCatalog/dataMapper.html) Мартина Фаулера (Martin Fowler) (вместо более популярного шаблона Active Record, http://martinfowler.com/eaaCatalog/activeRecord.html). В отличие от остальных библиотек, SQLAlchemy предоставляет не только уровень ORM, но и обобщенный API (который называется уровнем Core), предназначенный для написания кода без SQL. Слой ORM находится выше уровня Core, использующего объекты, которые непосредственно соотносятся с лежащей в его основе базой данных. Пользователь должен явно соотносить эти объекты и ORM, поэтому для начала работы потребуется написать больше кода (это может быть сложно для тех, кто только приступил к работе с реляционными базами данных — создавать объекты можно только явно).

SQLAlchemy может работать на Jython и PyPy и поддерживает версии Python от 2.5 до самой свежей. В следующих фрагментах кода показано, что нужно сделать, чтобы создать объекты с отношением «многие-ко-многим». Мы создадим три объекта на уровне ORM: `Customer` (Покупатель), `Cheese` (Сыр) и `Purchase` (Покупка). Один покупатель может сделать много покупок (отношение «многие-к-одному»), а в одной покупке может содержаться множество видов сыра (отношение «многие-ко-многим»). Мы приводим этот пример для того, чтобы показать несоотнесенную таблицу `purchases_cheeses` (ей не нужно находиться в ORM, поскольку она нужна только для связи между видами сыра и покупками).

Другие ORM создали бы эту таблицу за кулисами — в этом заключается одно из самых заметных различий между SQLAlchemy и другими библиотеками:

```
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy import Column, Date, Integer,
     String, Table, ForeignKey
from sqlalchemy.orm import relationship

Base = declarative_base()  ❶

class Customer(Base):  ❷
    __tablename__ = 'customers'
    id = Column(Integer, primary_key=True)
    name = Column(String, nullable=False)
    def __repr__(self):
       return "<Customer(name='%s')>" % (self.name)
```

```
purchases_cheeses = Table(  ❸
    'purchases_cheeses', Base.metadata,
    Column('purch_id', Integer, ForeignKey('purchases.id',
           primary_key=True)),
    Column('cheese_id', Integer, ForeignKey('cheeses.id',
           primary_key=True))
)

class Cheese(Base):  ❹
    __tablename__ = 'cheeses'
    id = Column(Integer, primary_key=True)
    kind = Column(String, nullable=False)
  purchases = relationship(  ❺
        'Purchase', secondary='purchases_cheeses',
        back_populates='cheeses'  ❻
    )
    def __repr__(self):
       return "<Cheese(kind='%s')>" % (self.kind)

class Purchase(Base):
    __tablename__ = 'purchases'
    id = Column(Integer, primary_key=True)
    customer_id = Column(Integer, ForeignKey('customers.id',
                         primary_key=True))
    purchase_date = Column(Date, nullable=False)
    customer = relationship('Customer')
    cheeses = relationship(  ❼
        'Cheese', secondary='purchases_cheeses',
        back_populates='purchases'
    )
    def __repr__(self):
       return ("<Purchase(customer='%s', dt='%s')>" %
                (self.customer.name, self.purchase_date))
```

❶ *Декларативный базовый объект* — это метакласс[1], который перехватывает создание каждой таблицы из ORM и определяет соответствующую таблицу на уровне Core.

❷ Объекты на уровне ORM наследуют от декларативного базового объекта.

❸ Это *несоотнесенная таблица* на слое Core; это не класс, он не наследуется у декларативного базового объекта, соответствует таблице `purchases_cheeses` в базе данных и нужен для того, чтобы предоставить соотношение «многие-ко-многим» между сырами и идентификаторами покупок.

❹ Сравните ее с соотнесенной таблицей `Cheese` на уровне ORM. За кулисами таблица `Cheese.__table__` создается на основном слое. Она будет соответствовать таблице базы данных `cheeses`.

[1] Метаклассы для Python хорошо объясняются на Stack Overflow.

❺ Это отношение явно показывает отношение между соотнесенными классами `Cheese` и `Purchase`: они связаны друг с другом опосредованно с помощью вторичной таблицы `purchases_cheeses` (в противоположность непосредственному связыванию с помощью `ForeignKey`).

❻ `back_populates` добавляет слушателя событий, поэтому при добавлении нового объекта типа `Purchase` в `Cheese.purchases` объект типа `Cheese` также появится в `Purchase.cheeses`.

❼ Этот фрагмент — вторая половина реализации отношения «многие-ко-многим».

Таблицы явно созданы с помощью декларативного базового объекта:

```
from sqlalchemy import create_engine
engine = create_engine('sqlite://')
Base.metadata.create_all(engine)
```

А теперь взаимодействие, при котором используются объекты слоя ORM, выглядит так же, как и для других библиотек, имеющих ORM:

```
from sqlalchemy.orm import sessionmaker
Session = sessionmaker(bind=engine)
sess = Session()

leicester = Cheese(kind='Red Leicester')
camembert = Cheese(kind='Camembert')
sess.add_all((camembert, leicester))
cat = Customer(name='Cat')
sess.add(cat)
sess.commit()  ❶

import datetime
d = datetime.date(1971, 12, 18)
p = Purchase(purchase_date=d, customer=cat)
p.cheeses.append(camembert)  ❷
sess.add(p)
sess.commit()
```

❶ Вы должны явно вызывать метод `commit()` для отправки изменений в базу данных.

❷ Объекты, состоящие в отношениях «многие-ко-многим», не добавляются во время создания (их необходимо вручную вносить уже после создания).

Рассмотрим несколько примеров запросов:

```
>>> for row in sess.query(Purchase,Cheese).filter(Purchase.cheeses): ❶
...     print(row)
...
(<Purchase(customer='Douglas', dt='1971-12-17')>,
```

```
                    <Cheese(kind='Camembert')>)
(<Purchase(customer='Douglas', dt='1971-12-17')>,
                    <Cheese(kind='Red Leicester')>)
(<Purchase(customer='Cat', dt='1971-12-18')>,
                    <Cheese(kind='Camembert')>)
>>>
>>> from sqlalchemy import func
>>> (sess.query(Purchase,Cheese)  ❷
...     .filter(Purchase.cheeses)
...     .from_self(Cheese.kind, func.count(Purchase.id))
...     .group_by(Cheese.kind)
... ).all()
[('Camembert', 2), ('Red Leicester', 1)]
```

❶ Так создается отношение «многие-ко-многим» для таблицы `purchases_cheeses`, которая не соотносится с высокоуровневым объектом ORM.

❷ Этот запрос считает количество покупок каждого вида сыра.

Для того чтобы узнать больше, обратитесь к документации SQLAlchemy (http://docs.sqlalchemy.org/en/rel_1_0/).

Django ORM

Django ORM (https://docs.djangoproject.com/en/1.9/topics/db/) — это интерфейс, используемый Django для предоставления доступа к базе данных. Их реализация шаблона Active Record больше всего похожа реализацию шаблона ActiveRecord, написанную на Ruby on Rails.

Он тесно интегрирован с Django, поэтому вы обычно будете использовать его только при создании веб-приложения с помощью Django. Обратите внимание на руководство к Django ORM от Django Girls (http://bit.ly/django-orm-tutorial), если вы хотите отслеживать процесс сборки веб-приложения[1].

Если планируете попробовать поработать с Django ORM, не создавая веб-приложение целиком, скопируйте этот скелет проекта с GitHub, чтобы использовать *только* Django ORM (https://github.com/mick/django_orm_only), и следуйте приведенным инструкциям. Вы можете столкнуться с некоторыми изменениями для разных версий Django. Наш файл `settings.py` выглядит следующим образом:

```
# settings.py
DATABASES = {
    'default': {
```

[1] Django Girls (https://djangogirls.org/) — благотворительная организация, в которую входят отличные программисты. Предоставляет возможность бесплатного обучения Django в среде, дружелюбной для женщин всего мира.

```
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': 'tmp.db',
    }
}
INSTALLED_APPS = ("orm_only",)
SECRET_KEY = "A secret key may also be required."
```

Каждая абстрагированная таблица в Django ORM является подклассом объекта Django Model. Это выглядит так:

```
from django.db import models

class Cheese(models.Model):
    type = models.CharField(max_length=30)

class Customer(models.Model):
    name = models.CharField(max_length=50)

class Purchase(models.Model):
    purchase_date = models.DateField()
    customer = models.ForeignKey(Customer)  (1)
    cheeses = models.ManyToManyField(Cheese)  (2)
```

(1) Отношение `ForeignKey` обозначает отношение «многие-к-одному» — покупатель может сделать много покупок, но покупка связана с одним покупателем. Используйте `OneToOneField` для создания отношения «один-к-одному».

(2) Используйте `ManyToManyField` для записи отношения многие-ко-многим.

Далее нужно выполнить команду для сборки таблиц. Активизировав виртуальную среду, в командной строке перейдите в каталог, где находится файл `manage.py`, и введите следующий код:

```
(venv)$ python manage.py migrate
```

После создания таблиц следует добавить данные в базу. Без использования метода `instance.save()` данные из новой строки не попадут в базу данных:

```
leicester = Cheese.objects.create(type='Red Leicester')
camembert = Cheese.objects.create(type='Camembert')
leicester.save()  (1)
camembert.save()

doug = Customer.objects.create(name='Douglas')
doug.save()

# Добавим время покупки
import datetime
now = datetime.datetime(1971, 12, 18, 20)
day = datetime.timedelta(1)
```

```
p = Purchase(purchase_date=now - 1 * day, customer=doug)
p.save()
p.cheeses.add(camembert, leicester)  ❷
```

❶ Чтобы попасть в базу данных и чтобы вы могли ссылаться на другие объекты, объекты должны быть сохранены.

❷ Вы должны добавлять объекты, состоящие в отношениях «многие-ко-многим», отдельно.

Создание запросов с помощью ORM в Django выглядит так:

```
# Фильтр для всех покупок, которые произошли за последние семь дней:
queryset = Purchase.objects.filter(purchase_date__gt=now - 7 * day) ❶

# Показываем, кто покупал сыры,
  а также разновидности этих сыров в наборе запросов:
for v in queryset.values('customer__name', 'cheeses__type'):  ❷
    print(v)

# Объединим покупки по типу сыров:
from django.db.models import Count
sales_counts = (  ❸
    queryset.values('cheeses__type')
    .annotate(total=Count('cheeses'))  ❹
    .order_by('cheeses__type')
)
for sc in sales_counts:
print(sc)
```

❶ В Django оператор для фильтрации (`gt`, greater than — «больше») добавляется после двойного нижнего подчеркивания к атрибуту таблицы `purchase_date` (Django анализирует его за кулисами).

❷ Двойное нижнее подчеркивание после идентификатора внешнего ключа предоставит доступ к атрибуту соответствующей таблицы.

❸ В случае если вы не видели нотации, можете поместить в скобки длинное выражение и разбить его на несколько строк для удобочитаемости.

❹ Блок `annotate` набора запросов добавляет дополнительные поля к каждому результату.

peewee

Основная цель peewee (http://docs.peewee-orm.com/en/latest/) — предоставить тем, кто знает SQL, легковесный способ взаимодействия с базой данных. Что вы видите, то и получаете (вы не будете разрабатывать вручную верхний уровень, который создает абстрактную структуру таблиц за кулисами, как SQLAlchemy;

библиотека также не будет волшебным образом создавать нижний уровень под вашими таблицами вроде Django ORM). Ее цель заключается в том, чтобы заполнить другую нишу, — выполнять небольшое количество задач, но работать быстро, просто и по-питонски.

Здесь нет почти ничего «волшебного» за исключением создания первичных ключей для таблиц в том случае, если этого не сделал пользователь. Вы можете создать таблицу следующим образом:

```
import peewee
database = peewee.SqliteDatabase('peewee.db')

class BaseModel(peewee.Model):
    class Meta: (1)
        database = database (2)

class Customer(BaseModel):
    name = peewee.TextField() (3)

class Purchase(BaseModel):
    purchase_date = peewee.DateField()
    customer = peewee.ForeignKeyField(Customer,
               related_name='purchases') (4)

class Cheese(BaseModel):
    kind = peewee.TextField()

class PurchaseCheese(BaseModel):
    """Для отношения «многие-ко-многим»."""
    purchase = peewee.ForeignKeyField(Purchase)
    cheese = peewee.ForeignKeyField(Cheese)

database.create_tables((Customer, Purchase, Cheese, PurchaseCheese))
```

(1) peewee хранит детали конфигурации в пространстве имен `Meta`, эта идея заимствована у Django.

(2) Связываем каждый объект типа `Model` с базой данных.

(3) Первичный ключ будет добавлен неявно, если вы не сделаете этого самостоятельно.

(4) Эта строка добавляет атрибут `purchases` к записям `Customer` для упрощения доступа, но ничего не делает с таблицами.

Инициализируем данные и добавляем их в базу данных за один шаг с помощью метода `create()` или же сначала инициализируем их, а затем добавим (существуют настройки конфигурации для управления автоматической отправкой транзакций и вспомогательными программами для них). Это делается за один этап:

```
leicester = Cheese.create(kind='Red Leicester')
camembert = Cheese.create(kind='Camembert')
cat = Customer.create(name='Cat')
import datetime
d = datetime.date(1971, 12, 18)

p = Purchase.create(purchase_date=d, customer=cat)  ❶
PurchaseCheese.create(purchase=p, cheese=camembert)  ❷
PurchaseCheese.create(purchase=p, cheese=leicester)
```

❶ Добавьте объект (вроде `cat`) — и peewee будет использовать его первичный ключ.

❷ В отношении «многие-ко-многим» нет ничего волшебного — просто добавьте новые записи вручную.

Пример запроса выглядит так:

```
>>> for p in Purchase.select().where(Purchase.purchase_date > d - 1 * day):
...     print(p.customer.name, p.purchase_date)
...
Douglas 1971-12-18
Cat 1971-12-19
>>>
>>> from peewee import fn
>>> q = (Cheese
...     .select(Cheese.kind, fn.COUNT(Purchase.id).alias('num_purchased'))
...     .join(PurchaseCheese)
...     .join(Purchase)
...     .group_by(Cheese.kind)
...  )
>>> for chz in q:
...     print(chz.kind, chz.num_purchased)
...
Camembert 2
Red Leicester 1
```

Вам доступна коллекция надстроек (https://peewee.readthedocs.org/en/latest/peewee/playhouse.html#playhouse), содержащая продвинутую поддержку транзакций[1], поддержку пользовательских функций, которые могут получать данные и выполнять их обработку до помещения в хранилище (например, сжатие или хэширование).

PonyORM

PonyORM (http://ponyorm.com/) применяет другой подход к грамматике запросов: вместо написания языка, похожего на SQL, или булевых выражений он использует синтаксис генератора Python. Также он имеет графический редактор схем, который может генерировать сущности PonyORM. Поддерживает Python версий 2.6+ и 3.3+.

[1] Контексты транзакций позволяют отменять выполнение, если на промежуточном шаге возникает ошибка.

Для того чтобы синтаксис оставался интуитивно понятным, Pony требует, чтобы все отношения между таблицами работали в обоих направлениях — все связанные таблицы должны явно ссылаться друг на друга, например так:

```
import datetime
from pony import orm

db = orm.Database()
db.bind('sqlite', ':memory:')

class Cheese(db.Entity): ❶
    type = orm.Required(str) ❷
    purchases = orm.Set(lambda: Purchase) ❸

class Customer(db.Entity):
    name = orm.Required(str)
    purchases = orm.Set(lambda: Purchase) ❹

class Purchase(db.Entity):
    date = orm.Required(datetime.date)
    customer = orm.Required(Customer) ❺
   cheeses = orm.Set(Cheese) ❻

db.generate_mapping(create_tables=True)
```

❶ В базе данных Pony с помощью сущности `Entity` сохраняется состояние объекта, с ее помощью соединяются база данных и сам объект.

❷ Pony использует стандартные типы Python для определения типа столбца — от `str` до `datetime.datetime`, в дополнение к определенным пользователем сущностям вроде `Purchase`, `Customer` и `Cheese`.

❸ Здесь используется `lambda: Purchase`, потому что `Purchase` еще не определен.

❹ `orm.Set(lambda: Purchase)` — первая половина определения отношения «один-ко-многим» между `Customer` и `Purchase`.

❺ `orm.Required(Customer)` — вторая половина отношения «один-ко-многим» между `Customer` и `Purchase`.

❻ Отношение `orm.Set(Cheese)`, объединенное с `orm.Set(lambda: Purchase)` на шаге (3), определяет отношение «многие-ко-многим».

После того как мы определили сущности для данных, создание объекта будет выглядеть как и в других библиотеках. Сущности создаются на лету и отправляются с помощью вызова `orm.commit()`:

```
camembert = Cheese(type='Camembert')
leicester = Cheese(type='Red Leicester')
cat = Customer(name='Cat')
```

```
doug = Customer(name='Douglas')
d = datetime.date(1971, 12, 18)
day = datetime.timedelta(1)
Purchase(date=(d - 1 * day), customer=doug, cheeses={camembert, leicester})
Purchase(date=d, customer=cat, cheeses={camembert})
orm.commit()
```

Запросы в Pony действительно выглядят так, будто написаны на чистом Python:

```
yesterday = d - 1.1 * day
for cheese in (
        orm.select(p.cheeses for p in Purchase
        if p.date > yesterday) ❶
    ):
    print(cheese.type)

for cheese, purchase_count in (
        orm.left_join((c, orm.count(p)) ❷
        for c in Cheese
        for p in c.purchases)
    ):
    print(cheese.type, purchase_count)
```

❶ Так выглядит запрос, созданный с помощью синтаксиса генератора для Python.

❷ Функция `orm.count()` объединяет объекты путем подсчета.

SQLObject

SQLObject (http://www.sqlobject.org/) (выпущен в октябре 2002 года) — самый старый ORM в нашем списке. Его реализация шаблона Active Record, а также оригинальная идея перегрузки стандартных операторов (вроде `==`, `<`, `<=` и т. д.) как способа абстрагирования некоторой логики SQL в Python, которая теперь реализована почти во всех библиотеках ORM, сделали его весьма популярным.

Поддерживает множество баз данных (распространенные системы вроде MySQL, Postgres и SQLite и более экзотические вроде SAP DB, SyBase и MSSQL), но в данный момент — только Python 2.6 и Python 2.7. Его все еще активно сопровождают, но он становится менее распространенным по мере использования SQLAlchemy.

Records

Records (https://github.com/kennethreitz/records) — это минималистичная библиотека SQL, разработанная для отправки необработанных запросов SQL в разные базы данных. Представляет собой объединенные Tablib и SQLAlchemy, для которых написали хороший API и приложение командной строки (ведет себя как клиент SQL, способный выводить YAML, XLS и другие форматы Tablib). Records не со-

бирается заменять библиотеки ORM; обычно он используется для выполнения запросов к базе данных и создания отчетов (например, ежемесячных отчетов в виде электронной таблицы, куда сохраняются последние данные о продажах). Данные могут быть использованы в программе или импортированы в один из многих полезных форматов:

```
>>> import records
>>> db = records.Database('sqlite:///mydb.db')
>>>
>>> rows = db.query('SELECT * FROM cheese')
>>> print(rows.dataset)
name         |price
-------------|-----
red leicester|1.0
wensleydale  |2.2
>>>
>>> print(rows.export('json'))
[{"name": "red leicester", "price": 1.0}, {"name": "wensleydale", "price": 2.2}]
```

Records предлагает инструмент для командной строки, который экспортирует данные с помощью SQL:

```
$ records 'SELECT * FROM cheese' yaml --url=sqlite:///mydb.db
- {name: red leicester, price: 1.0}
- {name: wensleydale, price: 2.2}
$ records 'SELECT * FROM cheese' xlsx --url=sqlite:///mydb.db  > cheeses.xlsx
```

Библиотеки для работы с базами данных NoSQL

Существует целая вселенная баз данных not only SQL («не только SQL») — это понятие применимо к любой базе данных, не являющейся традиционной. Если вы заглянете в PyPI, то можете запутаться, поскольку увидите несколько десятков пакетов Python со схожими именами.

Мы рекомендуем искать сведения о том, какая библиотека больше всего подходит для продукта, на основном сайте проекта для Python (например, поищите в Google «Python site:vendorname.com»). Большая часть библиотек предоставляет Python API и руководство для быстрого старта. Рассмотрим несколько примеров.

- *MongoDB* — это распределенное хранилище документов. Вы можете рассматривать его как гигантский словарь Python (может находиться в кластере), имеющий собственный фильтр и язык запросов. Для получения API для Python обратитесь к странице https://docs.mongodb.com/getting-started/python/.
- *Cassandra* — это распределенное хранилище таблиц. Предоставляет возможность быстрого поиска и может работать с широкими таблицами, но не предназначено для выполнения объединений — его функция заключается в том, чтобы

иметь дубликаты представлений для данных, ключи для которых содержатся в разных столбцах. Для получения более подробной информации об API для Python обратитесь к странице http://www.planetcassandra.org/apache-cassandra-client-drivers/.

- *HBase* — это распределенное хранилище столбцов (в этом контексте «хранилище для столбцов» означает, что данные хранятся в виде <идентификатор строки, имя столбца, значение>, что позволяет работать с очень разреженными массивами вроде наборов данных, получаемых от ссылок `from` и `to` для сайтов Всемирной паутины). Хранилище создано на основе распределенной файловой системы Hadoop. Для получения более подробной информации об API для Python обратитесь к странице https://hbase.apache.org/supportingprojects.html.
- *Druid* (http://druid.io/) — это распределенное хранилище столбцов, предназначенное для сбора (и опционального объединения перед сохранением) данных о событиях (в этом контексте «хранилище столбцов» означает, что столбцы можно упорядочить и отсортировать, а затем хранилище может быть сжато для получения более высокой скорости ввода/вывода и меньшего отпечатка). По ссылке https://github.com/druid-io/pydruid вы можете найти API для Python на GitHub.
- *Redis* — это распределенное хранилище, размещающее в памяти данные в формате «ключ-значение». Идея в том, чтобы снизить задержку, отказавшись от выполнения операций чтения с диска/записи на диск. Например, вы можете сохранять результаты выполнения частых запросов для более быстрого поиска в Сети. По адресу http://redis.io/clients#python приводится список клиентов Python для Redis, который указывает, что предпочитаемым интерфейсом является redis-py, а по ссылке https://github.com/andymccurdy/redis-py вы можете найти страницу redis-py.
- *Couchbase* (http://www.couchbase.com/) — еще одно распределенное хранилище документов, его API больше похож на SQL (по сравнению с API для MongoDB, который больше похож на JavaScript). По ссылке http://developer.couchbase.com/documentation/server/current/sdks/python-2.0/introduction.html вы можете найти Python SDK для Couchbase.
- *Neo4j* — база данных графов, предназначенная для хранения объектов, связанных подобием графов. По ссылке http://neo4j.com/developer/python/ вы можете найти руководство по Neo4j для Python.
- *LMDB (Lightning Memory-mapped Database om Symas)* (https://symas.com/products/lightning-memory-mapped-database/) — база данных, хранящая данные в формате «ключ-значение» в файле, отображаемом в памяти. Это означает, что файл необязательно читать с самого начала для того, чтобы дойти до того места, где хранятся данные, поэтому его производительность равна производительности хранилища в памяти. Привязки для Python находятся в библиотеке lmdb (https://lmdb.readthedocs.io/).

Приложение. Дополнительная информация

Сообщество Python

В глобальное сообщество пользователей Python входит немало доброжелательных людей.

BDFL

Гвидо ван Россума (Guido van Rossum), создателя Python, зачастую называют BDFL (Benevolent Dictator for Life — великодушный пожизненный диктатор).

Python Software Foundation

Миссия *Python Software Foundation (PSF)* — продвигать, защищать и развивать язык программирования Python, а также поддерживать и способствовать росту международного сообщества программистов Python. Для того чтобы узнать больше, обратитесь к основной странице PSF http://www.python.org/psf/.

PEP

PEP расшифровывается как *Python Enhancement Proposal* (предложение по улучшению Python). В таких протоколах описываются изменения в самом Python, а также в его стандартах. Те, кому интересно изучать историю Python или сам проект языка, найдут эти протоколы довольно любопытными (даже те из них, которые в итоге были отклонены). Существует три разновидности протоколов, они определены в PEP 1 (https://www.python.org/dev/peps/pep-0001).

- *Стандарты*. Описывают новую функциональность или реализацию.
- *Информационные протоколы*. Описывают проблемы проектов, общие положения или содержат информацию, полезную для сообщества.
- *Процессы*. Подобные протоколы описывают процессы, связанные с Python.

Конференции Python

Крупными событиями в жизни сообщества Python являются конференции разработчиков. Две наиболее заметные — PyCon (проводится в США) и EuroPython (проводится в Европе). Полный список конференций см. по ссылке http://www.pycon.org/.

Notable-протоколы

Существует несколько обязательных к прочтению протоколов.

- PEP 8 — руководство по стилю для кода Python (https://www.python.org/dev/peps/pep-0008). Прочтите его полностью. И следуйте ему. Инструмент pep8 вам поможет (https://pypi.python.org/pypi/pep8).
- PEP 20 — «Дзен Питона» (https://www.python.org/dev/peps/pep-0020). PEP 20 представляет собой список из 19 утверждений, которые кратко описывают философию, лежащую в основе Python.
- PEP 257 — соглашения для строк документации (https://www.python.org/dev/peps/pep-0257). PEP 257 содержит руководство по семантике и соглашения, связанные со строками документации.

По адресу http://www.python.org/dev/peps/ вы можете узнать еще больше.

Отправка PEP

Новые протоколы будет рассматривать сообщество — и после обширных дискуссий они будут приняты или отклонены. На рис. A.1 показано, что происходит, когда кто-то отправляет черновую версию протокола.

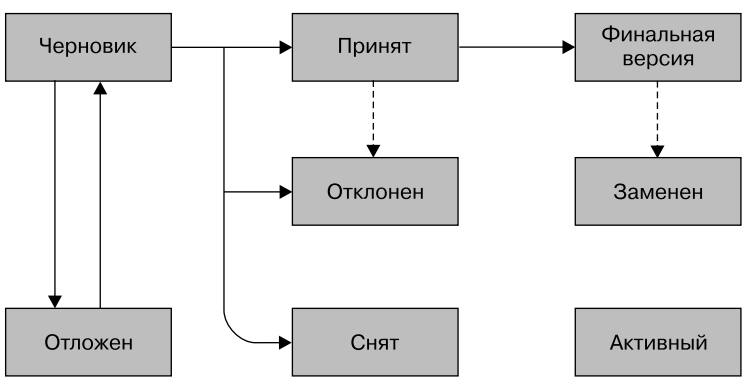

Рис. A.1. Обзор процесса рассмотрения протокола

Пользовательские группы Python

Пользовательские группы — это места, где разработчики Python встречаются лично для того, чтобы выступить с докладом или поговорить об интересующих их аспектах Python. Список локальных пользовательских групп см. в «Википедии»: http://wiki.python.org/moin/LocalUserGroups.

Обучение Python

Рассмотрим справочные материалы, сгруппированные по уровню сложности и областям применения.

Для начинающих

- *The Python Tutorial*. Официальное руководство по Python (http://docs.python.org/tutorial/index.html). Рассматриваются все основы, предлагается «экскурсия» по языку и стандартной библиотеке. Рекомендуется для тех, кому нужно руководство по языку для быстрого старта.
- *Python for Beginners*. Руководство (http://thepythonguru.com/) предназначено для начинающих программистов. Подробно рассматриваются многие концепции Python. Вы научитесь продвинутым конструкциям Python вроде лямбда-выражений и регулярных выражений. Руководство заканчивается статьей How to access a MySQL db using Python («Как получить доступ к базе данных MySQL с помощью Python»).
- *Learn Python*. Это интерактивное руководство (http://www.learnpython.org/) — легкий способ познакомиться с Python. Использует подход, реализованный на популярном сайте Try Ruby, — этот ресурс представляет собой интерактивный интерпретатор Python, встроенный в сайт, что позволяет пройти все уроки, не устанавливая Python на своем компьютере.
- *Python for You and Me*. Эта книга (http://pymbook.readthedocs.org/) — отличный ресурс для изучения всех аспектов языка; подходит для всех, кто предпочитает учиться по традиционной книге, а не по руководству.
- *Online Python Tutor*. Сайт http://pythontutor.com/ предлагает визуальное пошаговое представление того, как работает ваша программа. Python Tutor помогает пользователям преодолеть фундаментальный барьер, мешающий изучению программирования, показывая, что происходит, когда компьютер выполняет каждую строку исходного кода.
- *Invent Your Own Computer Games with Python*. Книга (http://inventwithpython.com/) предназначена для тех, у кого нет опыта программирования. Каждая глава содержит исходный код игры, и эти примеры программ используются для демонстрации концепций программирования, что помогает читателю понять, как программы «выглядят».

- *Hacking Secret Ciphers with Python.* Книга (http://inventwithpython.com/hacking/) помогает новичкам освоить программирование на языке Python и шифрование. Содержит исходные коды разнообразных шифров, а также программы, которые могут их взломать.
- *Learn Python the Hard Way*. Отличное руководство по Python для начинающих (http://learnpythonthehardway.org/book/). Рассматриваются многочисленные темы — от создания консольного приложения hello world до работы с Сетью.
- *Crash into Python.* Сайт http://stephensugden.com/crash_into_python/, также известный как Python for Programmers with 3 Hours, предлагает разработчикам, имеющим опыт работы с другими языками, экспресс-курс по Python.
- *Dive Into Python 3.* Эта книга (http://www.diveintopython3.net/) подходит для тех, кто готов окунуться в работу с Python 3. Она пригодится, если вы переходите с Python 2 на Python 3 или если у вас уже есть опыт программирования на других языках.
- *Think Python: How to Think Like a Computer Scientist.* Эта книга (http://greenteapress.com/thinkpython/html/index.html) пытается дать введение в основные концепции информатики с помощью языка Python. Идея создания этой книги заключается в том, чтобы предоставить книгу, содержащую множество упражнений, минимальное количество жаргонных выражений, а также раздел, посвященный отладке, в каждой главе. В ней рассматривается различная функциональность, доступная в Python, а также разные шаблоны проектирования и приемы хорошего тона.

  Книга также включает несколько примеров, позволяющих читателю исследовать темы, рассмотренные в книге, более подробно, применив их к примерам из реального мира. Среди примеров вы можете найти разработку графического пользовательского интерфейса и Markov Analysis.
- *Python Koans.* Это онлайн-руководство (http://bitbucket.org/gregmalcolm/python_koans) является версией для Python популярного инструмента Ruby Koans от Edgecase. Интерактивное руководство работает в командной строке, позволяет освоить базовые концепции Python с помощью тестов (http://en.wikipedia.org/wiki/Test-driven_development): путем исправления операторов контроля, которые дают сбой в тестовом сценарии, студенты последовательно продвигаются в освоении Python.

  Для тех, кто уже работал с языками программирования и разгадывал загадки самостоятельно, этот ресурс может показаться привлекательным. Новичкам в программировании может понадобиться еще один обучающий ресурс.
- *A Byte of Python*. Бесплатная вводная книга, которая обучает Python новичков (авторы подразумевают, что у читателей нет опыта программирования). Существуют отдельные версии для Python 2.x (http://www.ibiblio.org/swaroopch/byteofpython/read/) и для Python 3.x (http://swaroopch.com/notes/Python_en-Preface/)
- *Learn to Program in Python with Codecademy*. Этот курс Codecademy (http://www.codecademy.com/en/tracks/python) предназначен для тех, кто раньше не работал с Python.

Этот бесплатный интерактивный курс обучает основам (и выше) программирования на Python, при этом проверяя знания ученика по мере его продвижения по руководствам. В нем предоставляется встроенный интерпретатор, предназначенный для мгновенного получения обратной связи о вашей работе.

Средний уровень

Effective Python. В книге (http://www.effectivepython.com/) приводятся 59 способов повысить навык создания питонского кода. На 227 страницах представлен краткий обзор наиболее распространенных адаптаций, которые нужно выполнить для того, чтобы стать эффективным программистом Python среднего уровня.

Продвинутый уровень

- *Pro Python*. Книга (http://amzn.com/1430227575) предназначена для программистов Python среднего и продвинутого уровня, которые хотят понять, почему Python работает именно так, а не иначе, и как они могут перейти на новый уровень.
- *Expert Python Programming*. В этой книге (http://www.packtpub.com/expert-python-programming/book) показываются лучшие приемы программирования для Python. Она предназначена для более продвинутых пользователей. Книга начинается с тем вроде декораторов (в качестве примеров приводятся реализация кэша, прокси и менеджера контекста), порядка разрешения методов, использования метода `super()` и метапрограммирования, а также с приемов хорошего тона, описанных в PEP 8.

  В книге содержится подробный пример написания и выпуска пакета, который в итоге становится приложением, а также глава, посвященная использованию zc.buildout. Описываются правила хорошего тона вроде написания документации, разработки через тестирование, контроля версий, оптимизации и профилирования.
- *A Guide to Python's Magic Methods*. Этот полезный ресурс (http://www.rafekettler.com/magicmethods.html) представляет собой коллекцию статей Рафе Кеттлера (Rafe Kettler), в которых объясняются «волшебные методы» Python. Они окружены двойными нижними подчеркиваниями (например, `__init__`) и могут заставлять классы и объекты вести себя «волшебным» образом.

Для инженеров и ученых

- *Effective Computation in Physics*. Этот справочник Энтони Скопаца (Anthony Scopatz) и Кэтрин Д. Хафф (Kathryn D. Huff) (http://bit.ly/effective-computation-in-physics) предназначен для аспирантов, начинающих использовать Python в об-

ласти науки или инженерии. Включает в себя фрагменты кода, выполняющие поиск в файлах с помощью SED и AWK, а также содержит советы о том, как выполнить каждый шаг в цепи исследований — от сбора данных и анализа до публикации.

- *A Primer on Scientific Programming with Python.* В этой книге (http://bit.ly/primer-sci-pro-py) Ханса Петтера Лангтангена (Hans Petter Langtangen) в основном рассматривается использование Python для науки. Примеры подобраны из области математики и физики.
- *Numerical Methods in Engineering with Python.* В этой книге (http://bit.ly/numerical-methods-eng-py) Яна Куисалааса (Jaan Kiusalaas) делается акцент на современных численных методах и их реализации в Python.
- *Annotated Algorithms in Python: with Applications in Physics, Biology, and Finance.* Эта книга (http://amzn.com/0991160401) Массимо Ди Пьерро (Massimo Di Pierro) является инструментом обучения, предназначенным для демонстрации использованных алгоритмов, которые реализуются наиболее прямолинейным образом.

Дополнительные темы

- *Problem Solving with Algorithms and Data Structures.* В этой книге (http://www.interactivepython.org/courselib/static/pythonds/index.html) рассматривается набор структур данных и алгоритмов. Все представленные концепции подкреплены кодом Python и интерактивными фрагментами кода, которые вы можете запустить непосредственно из браузера.
- *Programming Collective Intelligence.* В этой книге (http://bit.ly/programming-collective-intelligence) приводится большое количество основных методов машинного обучения и дата майнинга. В ней допускаются некоторые вольности в том, что касается математической нотации. Она предназначена для того, чтобы объяснить логику, лежащую в основе этих методов, и показать способы реализации алгоритмов в Python.
- *Transforming Code into Beautiful, Idiomatic Python.* Видеоролик Реймонда Хеттингера (Raymond Hettinger) (http://bit.ly/hettinger-presentation) продемонстрирует, как наиболее эффективно использовать функциональность Python и улучшить код с помощью нескольких трансформаций: «Когда вы видите это, сделайте то».
- *Fullstack Python.* Сайт https://www.fullstackpython.com/ представляет собой полноценный ресурс, посвященный веб-разработке с помощью Python. Рассматривается широкий диапазон вопросов — от настройки веб-сервера до разработки фронтенда, а также выбор базы данных, оптимизация/масштабирование и многое другое. Из его названия следует, что на сайте вы можете узнать все, что нужно сделать, чтобы написать веб-приложение с нуля.

Справочный материал

- *Python in a Nutshell*. В этой книге (http://bit.ly/python-in-a-nutshell) рассматривается множество вопросов, связанных с кросс-платформенным использованием Python (от синтаксиса до встроенных библиотек), а также продвинутые темы вроде написания расширений на C.
- *The Python Language Reference*. Справочник по Python, доступный онлайн (http://docs.python.org/reference/index.html), в котором рассматриваются синтаксис и основы семантики языка.
- *Python Essential Reference*. Эта книга (http://www.dabeaz.com/per.html), написанная Дэвидом Бизли (David Beazley), представляет собой справочник по Python. В ней кратко объясняются основы языка и главные части стандартной библиотеки. В книге рассматриваются Python 3 и Python 2.6.
- *Python Pocket Reference*. Эта книга (http://bit.ly/python-pocket-reference), написанная Марком Лутцем (Mark Lutz), является простым справочником по основам языка, содержит описание наиболее часто используемых модулей и тулкитов. В книге рассматриваются Python 3 и Python 2.6.
- *Python Cookbook*. Эта книга (http://bit.ly/python-cookbook-3e), написанная Дэвидом Бизли (David Beazley) и Брайаном К. Джонсом (Brian K. Jones), содержит рецепты для разработчиков Python. В ней рассматриваются основы языка, а также задачи, которые приходится решать для многих прикладных областей.
- *Writing Idiomatic Python*. Книга написана Джеффом Наппом (Jeff Knupp), содержит наиболее популярные и важные идиомы Python. Каждая идиома представлена в виде рекомендации использовать какой-нибудь распространенный фрагмент кода, за которой следует объяснение, почему эта идиома так важна. В книге также содержатся два фрагмента кода для каждой идиомы — «вредный» способ писать код и «идиоматический». Для Python 2.7.3+ (http://amzn.com/1482372177) и для Python 3.3+ (https://amzn.com/B00B5VXMRG) изданы разные версии книги.

Документация

- *Официальная документация*. Официальную документацию для языка Python и его библиотеки вы можете найти по https://docs.python.org/2/ (для Python 2.x), а также здесь: https://docs.python.org/3/ (для Python 3.x).
- *Официальная документация по упаковке*. Самые последние инструкции по упаковке кода Python вы всегда можете найти в официальном руководстве по упаковке для Python (https://packaging.python.org/). И помните: существует testPyPI (https://testpypi.python.org/pypi), который позволит вам убедиться в том, что ваша упаковка работает корректно.
- *Read the Docs*. Это популярный проект сообщества, в котором хранится документация для ПО с открытым исходным кодом (https://readthedocs.org/). В нем содержится документация для многих модулей Python (как популярных, так и экзотических).
- *pydoc*. Это вспомогательная программа, которая устанавливается вместе с Python. Она позволяет вам быстро получать и искать документацию из вашей оболочки. Например, если вам нужно освежить в памяти принцип работы модуля time, получить документацию по нему можно, введя следующую команду в оболочке:

  ```
  $ pydoc time
  ```

  Эта команда эквивалентна открытию Python REPL и запуску такой команды:

  ```
  >>> help(time)*
  ```

Новости

Мы перечислим в алфавитном порядке наши любимые ресурсы, где можно прочесть новости о Python.

Название	**Описание**
/r/python	Сообщество на Reddit, посвященное Python, где пользователи публикуют новости, связанные с Python, а также оценивают их (http://reddit.com/r/python)
Import Python Weekly	Еженедельная рассылка, содержащая статьи, проекты, видеоролики и твиты, посвященные Python (http://www.importpython.com/newsletter/)
Planet Python	Агрегатор новостей о Python, получаемых от растущего количества разработчиков (http://planet.python.org/)
Podcast.__init__	Еженедельный подкаст, посвященный Python и людям, которые делают этот язык замечательным (http://podcastinit.com/)
Pycoder's Weekly	Бесплатная еженедельная рассылка для разработчиков Python от разработчиков Python (в ней приводятся интересные проекты, статьи, новости и предложения о работе) (http://www.pycoders.com/)
Python News	Новостной раздел официального сайта http://www.python.org/, посвященного Python. В нем кратко освещаются новости из сообщества, посвященного Python (http://www.python.org/news/)
Python Weekly	Бесплатная еженедельная рассылка, в которой приводятся избранные новости, статьи, новые релизы и предложения о работе, связанные с Python (http://www.pythonweekly.com/)
Talk Python to Me	Подкаст, посвященный Python и связанным с ним технологиям (http://talkpython.fm/)

Об авторах

Кеннет Ритц (Kenneth Reitz) — владелец продукта Python в Heroku и Fellow в Python Software Foundation. Он широко известен благодаря своим проектам с открытым исходным кодом, а особенно благодаря Requests: HTTP for Humans.

Таня Шлюссер (Tanya Schlusser) ухаживает за своей матерью, у которой болезнь Альцгеймера, работает независимым консультантом, используя данные метрик для принятия стратегических решений. Она потратила множество часов на обучение студентов и корпоративных клиентов работе с данными.

Об обложке

На обложке книги «Автостопом по Python» изображен индийский коричневый мангуст — небольшое млекопитающее из лесов Шри-Ланки и Юго-Восточной Индии. Он очень похож на короткохвостого мангуста (Herpestes brachyurus), обитающего на юго-востоке Азии, и может считаться его подвидом.

Индийский коричневый мангуст немного крупнее, чем мангусты других видов, он отличается остроконечным хвостом и мохнатыми задними лапами. Цвет меха варьируется от темно-коричневого (на теле) до черного (на лапах). Мангусты редко встречаются, поэтому можно предположить, что они ночные или сумеречные (активные на закате и рассвете) животные.

До недавнего времени данных об индийском коричневом мангусте было немного, а их популяция считалась малочисленной. С помощью новых методов наблюдения было обнаружено, что их популяция достаточно велика, особенно на юге Индии, поэтому их статус был повышен до Least Concern («наименьший риск»). Еще одна популяция индийских коричневых мангустов недавно была обнаружена на острове Вити-Леву (Фиджи).

Многие животные, изображенные на обложках книг издательства O'Reilly, находятся под угрозой исчезновения; все они важны для нашей планеты. Для того чтобы узнать, как вы можете помочь, посетите страницу animals.oreilly.com.

Изображение для обложки получено из Lydekker's Royal Natural History. На обложке использованы шрифты URW Typewriter и Guardian Sans. Для текста использован шрифт Minion Pro от Adobe, для заголовков — Myriad Condensed от Adobe, для кода — Ubuntu Mono от Dalton Maag.