

O'REILLY®



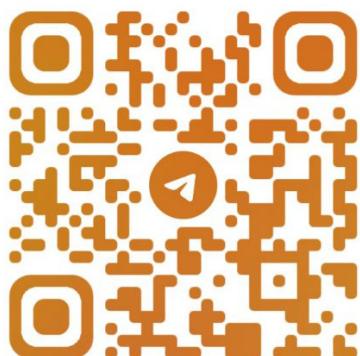
Авто- стопом по Python

THE HITCHHIKER'S GUIDE TO PYTHON

Кеннет Рейтц
Таня Шлюссер

The Hitchhiker's Guide to Python

Best Practices for Development



Kenneth Reitz and Tanya Schlusser

@CODELIBRARY_IT

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

Кеннет Рейтц, Таня Шлюссер

Автостопом по Python



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	<i>Ю. Сергиенко</i>
Руководитель проекта	<i>О. Сивченко</i>
Ведущий редактор	<i>Н. Гринчик</i>
Литературный редактор	<i>О. Андросик</i>
Художественный редактор	<i>С. Заматевская</i>
Корректоры	<i>О. Андреевич, Т. Курьянович</i>
Верстка	<i>А. Барцевич</i>

ББК 32.973.2-018.1

УДК 004.43

Рейтц К., Шлюссер Т.

P35 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.

Authorized Russian translation of the English edition of The Hitchhiker's Guide to Python.

© 2016 Kenneth Reitz, Tanya Schlusser

This translation is published and sold by permission of O'Reilly Media, Inc., which owns or controls all rights to publish and sell the same. All rights reserved.

ISBN 978-5-496-03023-6

© Перевод на русский язык ООО Издательство «Питер», 2017

© Издание на русском языке, оформление ООО Издательство «Питер», 2017

© Серия «Бестселлеры O'Reilly», 2017

Права на издание получены по соглашению с O'Reilly. Все права защищены. Никакая часть данной книги не может быть воспроизведена в какой бы то ни было форме без письменного разрешения владельцев авторских прав.

Информация, содержащаяся в данной книге, получена из источников, рассматриваемых издательством как надежные. Тем не менее, имея в виду возможные человеческие или технические ошибки, издательство не может гарантировать абсолютную точность и полноту приводимых сведений и не несет ответственности за возможные ошибки, связанные с использованием книги.

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

Введение 16

Условные обозначения 17

Благодарности 18

Часть I. Приступаем

Глава 1. Выбираем интерпретатор 20

Глава 2. Правильная установка Python 26

Глава 3. Ваша среда разработки 40

Часть II. Переходим к делу

Глава 4. Пишем отличный код 64

Глава 5. Читаем отличный код 116

Глава 6. Отправляем отличный код 195

Часть III. Руководство по сценариям

Глава 7. Взаимодействие с пользователем 214

Глава 8. Управление кодом и его улучшение 245

6 Краткое содержание

Глава 9. Программные интерфейсы	272
Глава 10. Манипуляции с данными	295
Глава 11. Хранение данных	308

Приложение. Дополнительная информация

Сообщество Python	326
Обучение Python	328
Документация	333
Новости	334
Об авторах	335
Об обложке	336

Оглавление

Введение	16
Условные обозначения	17
Благодарности	18
Часть I. Приступаем	
Глава 1. Выбираем интерпретатор	20
Python 2 против Python 3.....	20
Рекомендации	21
То есть... Python 3?	21
Реализации	22
CPython	22
Stackless.....	22
PyPy	23
Jython	23
IronPython	23
PythonNet	24
Skulpt	24
MicroPython	24
Глава 2. Правильная установка Python	26
Установка Python на Mac OS X	26
Setuptools и pip	28
virtualenv.....	29

Установка Python на Linux	30
SetupTools и pip	30
Инструменты разработки.....	31
virtualenv.....	32
Установка Python на Windows	33
SetupTools и pip	36
virtualenv.....	36
Коммерческие дистрибутивы Python	37
Глава 3. Ваша среда разработки	40
Текстовые редакторы	41
Sublime Text	42
Vim.....	43
Emacs.....	45
TextMate.....	46
Atom	46
Code.....	47
IDE	47
PyCharm/IntelliJ IDEA.....	49
Aptana Studio 3/Eclipse + LiClipse + PyDev.....	49
WingIDE	50
Spyder	51
NINJA-IDE.....	51
Komodo IDE.....	51
Eric (the Eric Python IDE).....	52
Visual Studio	52
Улучшенные интерактивные инструменты	53
IDLE	53
IPython.....	54
bpython	54
Инструменты изоляции	54
Виртуальные среды	55
pyenv	57
Autoenv	57

virtualenvwrapper	58
Buildout	59
Conda	60
Docker	61
 Часть II. Переходим к делу	
Глава 4. Пишем отличный код	64
 Стиль кода	
PEP 8	65
PEP 20 (также известный как «Дзен Питона»)	66
Общие советы	67
Соглашения.....	73
Идиомы	76
Распространенные подводные камни.....	80
Структурируем проект	83
Модули.....	83
Упаковка	87
Объектно-ориентированное программирование	88
Декораторы.....	90
Динамическая типизация	91
Изменяемые и неизменяемые типы	92
Зависимости, получаемые от третьей стороны	94
Тестирование вашего кода	95
Основы тестирования.....	97
Примеры	99
Другие популярные инструменты	103
Документация.....	105
Документация к проекту.....	106
Публикация проекта.....	106
Строки документации против блоковых комментариев.....	108
Журналирование	108

Журналирование для библиотеки.....	109
Журналирование для приложения.....	110
Выбираем лицензию	112
Лицензии	112
Доступные варианты	113
Лицензирование ресурсов	115
Глава 5. Читаем отличный код	116
Типичные функции	117
HowDoI.....	118
Читаем сценарий, состоящий из одного файла.....	118
Примеры из структуры HowDoI.....	122
Примеры из стиля HowDoI	123
Diamond	125
Читаем более крупное приложение.....	125
Примеры из структуры Diamond	131
Примеры из стиля Diamond	136
Tablib	139
Читаем небольшую библиотеку	139
Примеры из структуры Tablib.....	143
Примеры из стиля Tablib.....	151
Requests	154
Читаем более крупную библиотеку	154
Примеры из структуры Requests	157
Примеры из стиля Requests	162
Werkzeug	167
Читаем код инструментария	168
Примеры стиля из Werkzeug	176
Примеры структуры из Werkzeug	177
Flask	183
Читаем код фреймворка	184
Примеры стиля из Flask.....	190
Примеры структуры из Flask.....	192

Глава 6. Отправляем отличный код	195
Использование словаря и Concepts.....	196
Упаковываем код.....	197
Conda	197
PyPI.....	198
Замораживаем код	201
PyInstaller.....	203
cx_Freeze.....	205
py2app	207
py2exe.....	207
bbFreeze.....	208
Упаковка дистрибутивов в Linux	209
Исполняемые ZIP-файлы	210

Часть III. Руководство по сценариям

Глава 7. Взаимодействие с пользователем	214
Jupyter Notebook.....	214
Приложения командной строки	215
argparse	216
docopt	217
Plac	218
Click	219
Clint.....	221
cliff	222
Приложения с графическим интерфейсом	224
Библиотеки виджетов.....	224
Kivy	226
GTK+.....	228
wxWidgets	229
Objective-C	229
Разработка игр.....	230
Веб-приложения.....	231

Веб-фреймворки/микрофреймворки	232
Django.....	233
Flask.....	234
Tornado	234
Pyramid	235
Движки для веб-шаблонов	235
Jinja2.....	237
Chameleon.....	239
Mako	240
Развертывание веб-приложений.....	241
Хостинг	241
Веб-серверы.....	242
Серверы WSGI.....	243
 Глава 8. Управление кодом и его улучшение	 245
Непрерывная интеграция	245
Системное администрирование.....	246
Travis-CI	246
Jenkins	247
Buildbot	248
Автоматизация сервера.....	249
Salt.....	249
Ansible.....	250
Puppet.....	251
Chef	252
CFEngine.....	253
Наблюдение за системами и задачами	253
Psutil	253
Fabric	255
Luigi	256
Скорость.....	256
Многопоточность.....	259
Модуль multiprocessing	259
Subprocess.....	261

PyPy	261
Cython.....	262
Numba.....	265
Библиотеки для работы с GPU	266
Взаимодействие с библиотеками, написанными на C/C++/FORTRAN	267
C Foreign Function Interface.....	267
ctypes.....	268
F2PY.....	269
SWIG.....	270
Boost.Python.....	271
Глава 9. Программные интерфейсы	272
Веб-клиенты.....	273
API для сети	274
Анализ XML	276
Скрапинг сайтов	277
lxml	277
Сериализация данных.....	278
Pickle.....	279
Межъязыковая сериализация	279
Сжатие.....	281
Протокол буфера	281
Распределенные системы	281
Работа с сетью	282
Производительность сетевых инструментов из стандартной библиотеки Python.....	283
gevent	285
Twisted	285
PyZMQ.....	286
RabbitMQ.....	287
Шифрование	288
ssl, hashlib и secrets	290
pyOpenSSL.....	291
PyNaCl и libnacl.....	292

Cryptography.....	293
PyCrypto	293
bcrypt.....	294
Глава 10. Манипуляции с данными	295
Научные приложения	297
NumPy.....	297
SciPy.....	298
Matplotlib.....	298
Pandas.....	299
Scikit-Learn	299
Rpy2.....	299
decimal, fractions и numbers.....	299
SymPy	300
Манипуляции с текстом и его анализ.....	301
Инструменты для работы со строками стандартной библиотеки Python	301
nltk.....	302
SyntaxNet	305
Работа с изображениями	305
Pillow.....	305
cv2	306
Scikit-Image	307
Глава 11. Хранение данных	308
Структурированные файлы.....	308
Библиотеки для работы с базами данных	309
sqlite3.....	312
SQLAlchemy	313
Django ORM.....	316
peewee.....	318
PonyORM	320
SQLObject.....	322
Records.....	322
Библиотеки для работы с базами данных NoSQL	323

Приложение. Дополнительная информация

Сообщество Python	326
BDFL.....	326
Python Software Foundation	326
PEP	326
Конференции Python	326
Notable-протоколы	327
Отправка PEP	327
Пользовательские группы Python	327
Обучение Python	328
Для начинающих	328
Средний уровень	330
Продвинутый уровень.....	330
Для инженеров и ученых	330
Дополнительные темы.....	331
Справочный материал	332
Документация	333
Новости	334
Об авторах	335
Об обложке	336

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 (без пелляционное) руководство от специалистов, где рассматриваются популярные инструменты и лучшие практики на шего 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 главы 1 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 спрятаться (например, существует реализация MicroPython, 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 речь идет о 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, также внутри блоков, чтобы обратиться к элементам программы вроде переменных, функций и типов данных. Имена также выделены именами и расширения файлов.

Полужирный моноширинный шрифт

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

Курсивный моноширинный шрифт

Показывает текст, который должен быть изменен 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, и сколько мне известно, первая в своем роде: она создана одним втором (мной — Kenneth), большая часть содержимого предоставлена сотнями людей со всего света бесплатно. В истории человечества никогда не было такого количества технологий, которые позволили 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 — Даун (Dawn), Жасмин (Jasmine), Ник (Nick), Хезер (Heather), Николь (Nicole), Мег (Meg) и десятки людей, которые работали для того, чтобы эта книга стала еще лучше.

Часть I. Приступаем

Эт ч ать посвящен н стройке среды Python. Н с вдохновило руководство по Python для Windows втор Стю арт Элисс (Stuart Elliss) (<http://www.stuartellis.eu/articles/python-development-windows/>). Перв я ч ать состоит из следующих гл в.

- ❑ Гл в 1 «Выбир ем интерпрет тор». Мы сп вним Python 2 и Python 3, т кже р ссмотрим некоторые интерпрет торы в к честве льтерн тивы CPython.
- ❑ Гл в 2 «Пр вильн я уст новк Python». Мы пок жем, к к получить Python, т кже инструменты pip и virtualenv.
- ❑ Гл в 3 «В ш сред р зр ботки». Мы опишем н ши любимые текстовые ред кторы и IDE для р зр ботки 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.

Дел обстоят следующим образом:

- долгое время стандартом был Python 2.7;
- в Python 3 произошли значительные изменения в языке (этим немного недовольны некоторые разработчики¹);
- Python 2.7 будет получать обновления безопасности до 2020 года (<https://www.python.org/dev/peps/pep-0373/>);
- Python 3 продолжает развиваться, к этому происходило с Python 2 в последние годы.

Теперь вы понимаете, почему этот выбор не из простых.

¹ Если вы 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 в текстовом 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

Когда же придется, опытный разработчик скорее выберет Python 3. Но, если вы можете работать только с Python 2, вы все равно будете считаться пользователями Python. Вот наши рекомендации.

Используйте Python 3, если...

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

Используйте Python 2, если...

- вам нравится Python 2 и вспечали мысль, что в будущем вас ждет лишь Python 3;
- это повлияет на требования к системности вашего приложения¹;
- этого требует программное обеспечение, от которого вы зависите.

То есть... Python 3?

Если вы выбираете интерпретатор для Python и не имеете предубеждений, в most следует использовать новейший Python 3.x (в каждой версии появляются новые и улучшенные модули стандартных библиотек, а также исправления ошибок и проблем в безопасности). Прогресс не стоит на месте. Выбирайте Python 2, только если нет серьезных причин, например вы работаете с библиотекой, которая использует Python 2 и не имеет decks для 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 обработитесь к Python2orPython3 (<http://bit.ly/python2-or-python3>), где указываются причины обработной 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, остальное напишите 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, который может быть реализован множеством разных способов.

Разные реализации могут пригодиться для совместимости с другими библиотеками или же для небольшого ускорения. Библиотеки Python должны работать независимо от версии 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.

C^hython

CPython (<http://www.python.org/>) — это базовая реализация Python, написанная на языке C. Он компилирует код Python в промежуточный байт-код, который затем интерпретируется виртуальной машиной. CPython предоставляет высший уровень совместимости с кетами Python с модулями расширения, написанными на C¹.

Если вы пишете программы 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 — в нем 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 для языка будет работать со всеми библиотеками, которые может использовать CPython). Эта версия языка имеет отличие от базовой, отвязываящий интерпретатор Python от стека вызовов, что позволяет изменять порядок выполнения кода. Stackless вводит концепцию *taskletов*, которые могут обрабатывать функции и превращать их в «микропотоки» (они могут быть сери-

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

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

лизов ны и диск для последующего выполнения и пл ниров ния, по умолч нию выполняются по методу циклического перебор).

Библиотек greenlet (<http://greenlet.readthedocs.org/>) ре лизует т кую же функциональность по смене стек 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 ищет в себе цельность и максимум 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, PyPy3 (<http://pypy.org/compat.html>) и целенаправленный 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. Он может использовать библиотеки, написанные как для Python, так и с помощью .NET, а также предоставляет доступ к коду Python другим языкам фреймворка .NET.

Н дстройк Python Tools for Visual Studio (<http://ironpython.net/tools/>) интегрирует IronPython непосредственно в среду р з ботки Visual Studio, что дел ет эту ре - лиз цию иде льным выбором для р з ботчиков, использующих Windows.

IronPython поддержив ет Python 2.7 (<http://ironpython.codeplex.com/releases/view/81726>).

PythonNet

Python for .NET (<http://pythonnet.github.io/>) — это п кет, который предост вляет пр ктически бесшовную интегр цию н тивно уст новленного Python и общеязыковой среды выполнения .NET (Common Language Runtime (CLR)). Т кой под - ход противоположен подходу, которым пользуется IronPython. Это озн ч ет, что PythonNet и IronPython дополняют друг друг , не конкурируют.

В совокупности с Mono (<http://www.mono-project.com/>) PythonNet позволяет н тивным уст новк м Python в опер ационных систем х, отлич ющихся от Windows, н пример OS X и Linux, р бот ть с фреймворком .NET. Ре лиз ция может быть без конфликтов з пущен вместе с IronPython.

PythonNet поддержив ет версии от Python 2.3 до Python 2.7. Инструкции по уст - новке вы н ѹдете на стр нице <http://pythonnet.github.io/readme.html>.

Skulpt

Skulpt (<http://www.skulpt.org/>) — это ре лиз ция Python н JavaScript. Для нее еще не был полностью портирован ст нд ртн я библиотек CPython. Библиотек вклю - ч ет в себя модули math, random, turtle, image, unittest, т кже ч сти модулей time, urllib, DOM и т.е. Предн знчен для использов ния при обучении. Кроме того, есть возможность доб вить собственные модули (<http://bit.ly/skulpt-adding-module>).

Стоит упомянуть примеры ее применения — Interactive Python (<http://interactive-python.org/>) и CodeSkulptor (<http://www.codeskulptor.org/demos.html>).

Skulpt поддержив ет большую ч сть функцион льности версий Python 2.7 и Python 3.3. Для получения более подробной информ ции смотрите стр ницу ре лиз ции Skulpt на GitHub (<https://github.com/skulpt/skulpt>).

MicroPython

MicroPython (<https://micropython.org/>) — это ре лиз ция Python 3, оптимизирована для з пуск на микроконтроллере. Поддержив ет 32-битные процессоры ARM, имеющие на бор инструкций Thumb v2, т кие к к Cortex-M, широко используемые в дешевых микроконтроллер х. Включ ет в себя модули стандартной библиотеки Python (<http://bit.ly/micropython-library>), т кже несколько

библиотек, некоторых для MicroPython, которые отвечают за подробную информацию о плате и питании, доступ к сетям, а также модифицированную версию ctypes, оптимизированную для уменьшения размеров. Этот релиз не похож на Raspberry Pi (<https://www.raspberrypi.org/>) — поддерживая Debian или другую операционную систему, основную на C, на которой установлен Python. Платформа 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 покажем процесс установки CPython на платформах Mac OS X, Linux и Windows. Информация в разделах, посвященных инструментам управления (вроде Setuptools и pip), повторяется, поэтому можете сразу перейти к разделу, соответствующему вашей системы.

Если вы работаете в организации, которой я рекомендует 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

С момента поздней версии Mac OS X — El Capitan¹ — появляется с собственной реализацией Python 2.7.

Вместо этого нужно установить 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 внедрения включенных в совместные проекты). В этом разделе вы

¹ На момент написания книги.

узн ете больше о том, к к их уст н влив ть и использов ть. В ч стности, в м всегда следует уст н влив ть Setuptools, поскольку это зн чительно упрощ ет применение других сторонних библиотек Python.

Т версия Python, котор я пост вляется вместе с OS X, отлично подходит для обучения, но не годится для совместной р зр ботки. Эт версия может быть уст ревшей по спр внению с офици льной текущей версией, котор я счит ется ст бильной производственной версией¹. Поэтому, если вы хотите использозв ть Python только для того, чтобы пис ть сцен рии для себя, получ ть информ цию с с ютов или обр б тыв ть д нные, в м больше ничего не пон добится. Но если вы собираетесь вносить свой вкл д в проекты с открытым исходным кодом или р бот ть в ком нде с сотрудник ми, которые могут пользов ться другими опер - ционными систем ми (или пл нируете з ним ться этим в будущем²), выбир йте версию CPython.

Перед тем к к вы что-либо з грузите, прочтите следующие бз цы. До уст - новки Python в м нужно уст новить GCC. Для этого з грузите Xcode (<http://developer.apple.com/xcode/>), упрощенную версию Command-Line Tools (<https://developer.apple.com/downloads/>) (в м пон добится учетн я з пись Apple) или д же версию п кет 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 н то что OS X пост вляется с большим количеством утилит Unix, р зр ботчики, зн комые с систем ми Linux, обн руж т отсутствие одного ключевого компонент : подходящего менеджер п кетов. Эту брешь может з полнить инстру - мент Homebrew.

¹ У р зных людей р зные мнения н этот счет. Р елиз ции OS X Python могут р зли - ч ться. Они д же могут иметь р зные библиотеки, х р ктерные для OS X. Вы можете прочит ть небольшую ст тью, втор которой критикует н ши рекоменд ции, в блоге Stupid Python Ideas (<http://bit.ly/sticking-with-apples-python>). В ней подним ются логичные вопросы о конфликт х имен, которые возник ют, когд пользов тель переключ ется с CPython 2.7 для OS X н CPython 2.7, нез висимый от конечной ре лиз ции. Если для в с это может ст ть проблемой, используйте вирту льную среду. Или хотя бы ост вьте CPython 2.7 для OS X в том же месте, чтобы систем четко р бот л , уст - новите ст нд артный Python 2.7, ре лизов нный с помощью CPython, измените путь и никогда не пользуйтесь версией OS X. Д лее все будет р бот ть к к ч сы, включ я продукты, которым нужн версия Python для OS X.

² Честно говоря, лучший в ри нт — с с мого н ч л использозв ть либо Python 3, либо вирту льные среды, куд вы не будете уст н влив ть ничего, кроме virtualenv и, возможно, virtualenvwrapper в соответствии с совет ми Хинек Шл в к (Hynek Schlawack) (<https://hynek.me/articles/virtualenv-lives/>).

Чтобы уст новить Homebrew, откройте Terminal или в ш любимый эмулятор консоли для OS X и з пустите следующий код:

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

Сцен рий объяснил, к кие изменения он собир ется внести, и предупредит в с перед н ч лом процесс . К к только вы уст новите Homebrew, доб вьте к т лог Homebrew в н ч ло в шей переменной среды (окружения) PATH¹. Вы можете сдел ть это, доб вив следующую строку в нижнюю ч сть ф йл `~/.profile`:

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

Для уст новки Python з пустите в консоли следующую ком нду:

```
$ brew install python3
```

Для Python 2:

```
$ brew install python
```

По умолчанию Python будет уст новлен в к т лог `/usr/local/Cellar/python3/` или `/usr/local/Cellar/python/` с символьными ссылк ми² для интерпретатор по дресу `/usr/local/python3` или `/usr/local/python`. Если вы пл нируете использовать п р метр `--user` для ком нды `pip install`, то придется обойти б г, который к с ется `disutils` и конфигур ции Homebrew. Мы рекомендуем р бот ть с виртуальными сред ми, опис иными в подр зделе «virtualenv» д лее.

Setup tools и pip

Инструмент Homebrew уст новит Setuptools и pip. Исполняемые ф йлы, уст новленные 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 з грузить и уст новить любое совместимое³ ПО для Python по сети (обычно по Интернету), выполнив всего одну ком нду

¹ Это г р нтирует, что версия Python, которую вы собир етесь использовать, будет той, которую только что уст новил Homebrew (ост вил оригиналную версию Python нетронутой).

² Символьн я ссылк — это ук з тельн ре льное местоположение ф йл . Вы можете проверить, куд ведет ссылк , н бр в, н пример, `ls -l /usr/local/bin/python3` в ком ндной строке.

³ П кеты, которые миним льно совместимы с Setuptools, предост вляют досточно информ ции для библиотеки, чтобы он мог идентифициров ть и получить все з висимые п кеты. Для более подробной информ ции обр титесь к документ ции `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 тоже позволит вам добавить возможность устанавливать софт по сети для в ших собственных программ, написанных на Python, не затрачивая много усилий.

Команды `pip` для pip и команда `easy_install` для `Setuptools` – средства для установки и управления пакетами Python. Первую используете предпочтительнее, поскольку он тоже может удалять пакеты, ее сообщения об ошибке более понятны, что стичные установки пакетов невозможны (если процесс достиг сбоя, 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 – к тому же, содержащий все исполняемые файлы, которые будут использоваться пакеты, что могут помочь в проекту, написанному на Python.

Некоторые считают, что хорошим тоном является установка только лишь `virtualenv` и `Setuptools` и дальнейшая работа исключительно в виртуальной среде¹.

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

```
$ pip3 install virtualenv
```

Для Python 2:

```
$ pip install virtualenv
```

К тому же вы можете использовать `virtualenv` 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» глобально используется и мотивация описывая более подробно.

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

Установка Python на Linux

Ubuntu выпуск ется только с предуст новленной версией Python 3 (версия Python 2 доступн по ком нде `apt-get`), и чин я с версии Wily Werewolf (Ubuntu 15.10). Все подробности вы можете узи ть и стр нице 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 доступн только бл год ря менеджеру п кетов.

Больш я ч сть п р ллельных уст новок Python 2 и Python 3 созд ет символьную ссылку и интерпреторы Python 2 и Python 3. Если вы решите использовать Python 2, текущей рекоменд цией для Unix-подобных систем (см. Python Enhancement Proposal [PEP 394] (<https://www.python.org/dev/peps/pep-0394/>)) является необходимость явно ук зыв ть `python2` (н пример, `#!/usr/bin/env python2` в первой строке ф йл), не пол г ясь и то, что сред с м все сдел ет.

Несмотря и то что этого нет в PEP 394, теперь принято испльзов ть `pip2` и `pip3`, чтобы ук зыв ть и соответствующие уст новщики п кетов.

Setup tools и pip

Хотя `pip` доступен в в шей системе бл год ря уст новщику п кетов, для того чтобы г р итиров ть, что вы будете р бот ть с с мой новой версией, выполните следующие ш ги.

1. Для и ч л з грузите `get-pip.py`¹ (<https://bootstrap.pypa.io/get-pip.py>).
2. Д лее откройте оболочку, измените к т логи т к, чтобы они ук зыв ли и 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 ком нды т кже уст новят Setup tools.

С помощью ком нды `easy_install`, котор я доступн бл год ря Setup tools, вы можете з грузить и уст новить любое совместимое² ПО для Python по сети (обычно

¹ Для получения более подробной информ ции обр титесь к инструкции по уст новке `pip` (<https://pip.pypa.io/en/latest/installing.html>).

² П кеты, которые миним льно совместимы с Setup tools, предост вляют досточно информ ции для библиотеки, чтобы он мог идентифициров ть и получить все з висимые п кеты. З более подробной информ ции обр титесь к документ ции *Packaging and Distributing Python Projects*, PEP 302 и PEP 241.

по Интернету). Это также позволит вам добавить возможность установки ПО по сети для в ших собственных прогрессов, написанных на Python, не затрачивая много усилий.

Команды `pip` позволяет легко устанавливать пакеты Python и управлять ими. Рекомендуется использовать именно этот инструмент вместо `easy_install`, поскольку этот командой также может удалять пакеты, ее сообщения об ошибке более понятны, что статические установки пакетов невозможны (если процесс достигнет сбоя, все его результаты будут отменены).

Инструменты разработки

Практически каждый пользователь Python в какой-то момент будет интересоваться библиотеками, которые зависят от расширений, написанных на языке С. Возможно, в менеджере пакетов будет иметься функция сборки новых библиотек, поэтому вы можете сами проверить это (с помощью команды `yum search` или `apt-cache search`). Более новый формат `wheels` (<http://pythonwheels.com/>) (здесь скомпилированные пакетные файлы для платформы binary) дает возможность получить бинарные файлы непосредственно из PyPI с помощью команды `pip`. Если вы планируете в будущем создавать расширения на С, или если люди, поддерживающие вашу библиотеку, не создали решения с помощью `wheels` для вашей платформы, в этом понадобятся инструменты разработки для Python: различные библиотеки, написанные на С, команда компилятора GCC. Перечислим полезные пакеты, которые работают с библиотеками, написанными на С.

Пакеты для работы с конкуренцией:

- ❑ библиотеки для работы с потоками `threading` (<https://docs.python.org/3/library/threading.html>);
- ❑ библиотеки для обработки событий (Python 3.4+) `asyncio` (<https://docs.python.org/3/library/asyncio.html>);
- ❑ библиотеки, основанные на сопрограммах, `curio` (<https://curio.readthedocs.org/>);
- ❑ библиотеки для работы с сетями, основанные на сопрограммах, `gevent` (<http://www.gevent.org/>);
- ❑ управляемые событиями библиотеки для работы с сетями `Twisted` (<https://twistedmatrix.com/>).

Нучный анализ:

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

Интерфейс для работы с данными/базой данных:

- ❑ интерфейс для форматов HDF5 h5py (<http://www.h5py.org/>);
- ❑ драйвер для базы данных PostgreSQL Psycopg (<http://initd.org/psycopg/>);
- ❑ быстрая база 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.) Вместе же 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 в этом 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. Он 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 устновить пакет с помощью комманды `pip`. Запустите менеджер в командной строке консоли оболочки и добавьте путь к директории `--user`, чтобы устанавливать пакет локально для себя (не выполняя установку для всей системы):

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

Для Python 2:

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

К сожалению вы окажетесь в виртуальной среде, всегда сможете использовать комманду `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 впрочем используют расширения, написанные на C.

В большинстве форматов wheels бинарные файлы можно загружать из PyPI с помощью `pip` (если они существуют), поэтому работать с Python стало гораздо проще.

У вас есть две пути: использовать коммерческий дистрибутив (они представлены в разделе «Коммерческие дистрибутивы Python» далее) или CPython. Работать с дистрибутивом Anaconda гораздо проще, особенно если вы собираетесь с ним работать учеными расчетами. Пррактически каждый разработчик, применяющий Python для Windows (кроме тех, кто с mostостоятельно разрабатывает библиотеки для Python, основанные на C), порекомендует Anaconda. Но если вы собираетесь 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 CPython¹ (когда требуется интегрировать Python во фреймворк .NET). Однако, если вы только начинаете изучать Python, знание языка с этим интерпретатором, скорее всего, лучше отложить на будущее. На протяжении этой книги мы расскажем о CPython.

Со временем все больше пакетов, содержащих библиотеки, написанные на C, будут поддерживаться в формате wheels для PyPI. Их можно будет получить, воспользовавшись командой `pip`. Проблемы могут возникнуть, если у вас в 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 (последний разделится в подразделе «IronPython» раздела «Релелизации» главы 1).

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

- в м не нужны библиотеки Python, которые зависят от расширений, написанных на C;
- у вас есть компилятор для Visual C++ (не тот, что распространяется бесплатно);
- можете использовать MinGW;
- можете загружать binaryные файлы вручную¹, в тем случае новить их с помощью команда `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), выберите Anaconda².

Если же работаете в IDE (integrated development environments – интегрированные среды разработки), чей интерфейс в основном графический («укажите и щелкните»), или если Python – ваш первый язык программирования и в нем предстоит окунуться в первую установленную IDE, используйте Canopy.

Когда вся в нем уже применяет один из предложенных здесь вариантов, в м следует действовать также.

Чтобы установить стандартную версию 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 установите последнюю версию (либо же знаете, что в нем необходим 64-битный установщик³), можете воспользоваться ресурсом Python Releases for Windows (<https://www.python.org/downloads/windows/>) (там найдете необходимый в нем релиз).

¹ Вы должны знать хотя бы версию Python, которую планируете использовать, т.к. же требуемую зарядность. 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 binaryные файлы, – сайт ресурсов Кристофа Гольке (Christoph Gohlke) (<http://www.lfd.uci.edu/~gohlke/pythonlibs/>). Что касается scikit-learn, Карла Клэффнера (Carl Kleffner) выполняет сборку binaryных файлов с помощью MinGW (<https://pypi.anaconda.org/carlkl/simple/>) для подготовки к итоговому релизу для PyPI.

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

³ Это означает, что вы не 100 % уверены, что все необходимые DLL и другие файлы имеют 64-битную версию.

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

По умолчанию Python устанавливается в корневой лог, в котором которого встроен номер версии (например, версия Python 3.5 будет установлен в корневой лог `C:\Python35\`), поэтому у вас может быть несколько версий Python в 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 запуски разделены точкой с запятой в том числе, поэтому вы всегда сможете с помощью выбирать, которую именно копию Python следует запустить.

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

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

Вы можете сделать это, запустив в PowerShell² следующие команды:

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

Второй корневой лог (`Scripts`) получает файлы команд, когда устанавливается определенные пакеты (это очень полезное дополнение). В нем потребуется установить или конфигурировать что-то еще для применения Python.

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

¹ В переменной среды `PATH` перечислены все возможные корневые логи, в которых операционные системы будут искать исполняемые программы (например, Python и его зависимости 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 прочтите в статье «Using Windows PowerShell от компании Microsoft» (<http://bit.ly/using-windows-powershell>).

SetupTools и pip

MSI уст н влив ют Setuptools и pip вместе с Python, поэтому, если вы выполняете все рекомендации из этой книги и только что произвели уст новку, они у в с уже есть. В противном случае лучший способ их получить при работе с версией 2.7 – выполнить обновление до самой последней версии 15 (уст новщик спросит у в с, можно ли через писать существующую версию, — в м следует согласиться; релизы, имеющие одинковый вспомогательный номер версии, имеют и общую совместимость). Для Python 3 (версии 3.3 и выше) з грузите скрипт `get-pip.py`¹ (<https://bootstrap.pypa.io/get-pip.py>) и зпустите 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) згрузить и установить любое совместимое² ПО для Python, введя одну команду (`easy_install`). Это также позволит добавить возможность устанавливать софт по сети для встроенных программ, написанных с помощью Python, не затрачивая много усилий.

Команда `pip` для pip и команда `easy_install` для Setuptools являются инструментами для установки новой и управления проектами Python. Первую команду используйте предпочтительнее, поскольку она также может уделять некоторые ее сообщения об ошибке более понятны, а статичные устновки проектов 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. Он 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, он добавит его в начало конца строки, предшествующей собой переменную среды `PATH` — версия Python в новом каталоге будут обнаружены в первую очередь, в результате чего будут действовать 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, предстаивают досточно информацией для библиотеки, чтобы она могла идентифицировать и получить все зависимости пакеты. Более подробные сведения найдите 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 консоли PowerShell:

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

В OS X и Linux (поскольку Python предустановлен в системе) для использования системы или сторонним ПО) необходимо явно указать версию pip для Python 2 и Python 3. В Windows нам нужно этого делать, поэтому, когда мы говорим pip3, имеем в виду pip для пользователей Windows. Независимо от ОС, как только вы попадете в виртуальную среду, всегда можете использовать комманду pip — неважно, работете вы с Python 2 или Python 3 (это мы и будем делать на протяжении остальной части книги).

Коммерческие дистрибутивы Python

В ш отдел IT или препод в тель могут попросить в с уст новить коммерческий дистрибутив Python. Это необходимо, чтобы упростить р боту, которую должен выполнить орг низ ция, и поддержив ть ст бильную среду для нескольких поль зов телей. Все перечисленные здесь дистрибутивы предос тавляют ре лиз цию Python, написанную на C (CPython).

Н учный ред ктор первого черновик этой гл вы ск з л, что мы серьезно недо-
оценив ем неудобств , которые большинству пользов телей дост вляет обычн я
версия CPython на Windows: несмотря н существов ние форм т wheels, компи-
лиров ние и/или связыв ние внешних библиотек, н пис нных н C, предст вляет
трудность для всех, кроме опытных р зр ботчиков. Мы предпочит ем CPython, но,
если вы собир етесь пользов ться библиотек ми или п кет ми (не созд в ть их
или доб влять в них что-то свое), в м следует з грузить коммерческий дистрибу-
тив и просто н ч ть р бот ть (это особенно в жно, если вы р бот ете в Windows).
Когд з хотите внести свой вкл д в проекты с открытым исходным кодом, сможете
уст новить 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.

Кто 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 упр вления п кет ми используется conda от Continuum, но подойдет и pip. Дистрибутив можно з грузить с мостоятельно либо уст новить с сйт <https://anaconda.org/> в среде conda¹ (<http://bit.ly/intel-python-beta>).

Дистрибутив предост вляет стек SciPy и другие р спростр ненные библиотеки, перечисленные в сопроводительных документах (в формате PDF) (<http://bit.ly/intel-python-release-notes>). Пользователям Intel Parallel Studio XE доступен коммерческая поддержка, все остальные могут общаться на форумах. Позволяет в м 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 предост вляет множество из них скомпилированных и тематических бинарных файлов в своем каталоге бесплатно пользователей (<https://repo.continuum.io/pkgs/>). Он использует не pip, другой менеджер пакетов (conda), который также управляет виртуальными средами, но действует скорее как Buildout (рассматриваясь в подразделе «Buildout» раздела «Инструменты изоляции» главы 3), не как virtualenv (управляет библиотеками и другими внешними зависимостями для пользователей). 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 включает в себя функциональности. Если вам комфортно работать с команной строкой и интересуется 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 компоненты (связанные с лицензией и ПО с открытым исходным кодом, такими как 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 компонентов (связанных с лицензией и ПО с открытым исходным кодом) для более чем 7000 пакетов, расположенных в его каталоге пакетов (<https://code.activestate.com/ropm/>) (их можно получить с помощью инструмента pip, 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 epkg, который используется вместо pip для связи с текущим логом пакетов Canopy (<http://bit.ly/enthought-canopy>).

Enthought предоставляет бесплатные коммерческие лицензии студентам и разработчикам учреждений образовательного характера. 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 различными языками. Кроме других коммерческих дистрибутивов, в нем предусмотрены механизмы 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 предпочтем использовать Sublime Text в качестве редактора и PyCharm/IntelliJ IDEA (оба ссылаются в следующем разделе) в качестве IDE, но при этом понимаем, что лучший вариант зависит от решаемых 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, не компилируемый язык¹, поэтому мы не будем рассыпать эти вопросы. Но в главе 6 опишем, как использовать Setuptools для упаковки проектов и Sphinx для сборки документации.

Мы также не будем рассыпать системы контроля версий, поскольку они не зависят от языка, но программы, которые поддерживают различные 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 предсказано применение четырех вариантов, популярных в настоящее время: Subversion, Bazaar, Git и Mercurial.

Завершается глава таким обзором современных способов управления интерпретаторами для воссоздания различных ситуаций, которые могут возникнуть при развертывании, и это не кодировка.

¹ Если в самом понимании добиться создания 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 любой язык программ, который позволяет редактировать текст, один из выбора привильного 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 таблице 3.1 перечислены текстовые редакторы, которые нам нравятся (поубыванию предпочтения), и объясняется, почему разработчику следует выбрать именно этот редактор. Далее в главе кто-то рассмотрит каждого из них. В «Википедии» по адресу https://en.wikipedia.org/wiki/Comparison_of_text_editors приведен табличка, 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	<ul style="list-style-type: none"> Открытый API/бесплатный прообразный период. OS X, Linux, Windows 	<ul style="list-style-type: none"> Быстро работает и действует на маленький объем памяти. Способен работать с крупными файлами (> 2 ГБайт). Расширения написаны на Python
Vim	<ul style="list-style-type: none"> ПО с открытым исходным кодом/можно вносить пожертвования. OS X, Linux, Windows, Unix 	<ul style="list-style-type: none"> Встречается Vim. Предустановлен (во всяком случае Vim) на любой ОС кроме Windows. Может быть консольным приложением
Emacs	<ul style="list-style-type: none"> ПО с открытым исходным кодом/можно вносить пожертвования. OS X, Linux, Windows, Unix 	<ul style="list-style-type: none"> Встречается Emacs. Расширения написаны на Lisp. Может быть консольным приложением
TextMate	<ul style="list-style-type: none"> ПО с открытым исходным кодом/требует лицензии. Только для OS X 	<ul style="list-style-type: none"> 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	<ul style="list-style-type: none"> ПО с открытым исходным кодом/бесплатный. OS X, Linux, Windows 	<ul style="list-style-type: none"> Расширения написаны на JavaScript/HTML/CSS. Хорошая интеграция с GitHub
Code	<ul style="list-style-type: none"> Открытый API (в будущем)/бесплатный. OS X, Linux, Windows (но Visual Studio, соответствующая IDE, работает только в Windows) 	<ul style="list-style-type: none"> IntelliSense (автозависимые коды), предоставляемое VisualStudio. Хорошо подходит для разработчиков на Windows, поскольку поддерживает .Net, C# и F# <p>Примечание. Имеется недостаток: нельзя расширять (в будущем будет исправлен)</p>

Sublime Text

Мы советуем использовать текстовый редактор Sublime Text (<http://www.sublimetext.com/>) для написания кода, заметки 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, работает со своим API для расширения пакетов. Функция Projects (Проекты) позволяет пользователям 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, в меню достаточно PackageControl (<https://packagecontrol.io/installation>). Популярные пакеты включают SublimeLinter, интерфейс для работы с выбранными и установленными пользователям телем статическими проверками кода, Emmett для работы со сниппетами¹ для веб-разработчиков и Sublime SFTP для удаленного редактирования с помощью FTP.

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

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

Vim

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

Vim можно расширить с помощью vimscript — простого языка для написания скриптов. Можно использовать другие языки. Для включения возможности написания скриптов на Python устновите следующие файлы конфигурации при сборке исходного кода, написанные на C: `--enable-pythoninterp` и/или `--enable-python3interp`. После этого можете начать сборку исходников. Чтобы узнать, какие версии Python в наличии доступны, введите `:echo has("python")` или `:echo has("python3")`. Результат том будет 1, если выражение верно, и 0, если нет.

Vi (изначально 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, связанный с IT, в нем необходимо уметь работать с Vi¹. У Vim гораздо больше функций, чем у Vi, но он похож на предшественник настолько, что пользователь Vim спрятится с Vi.

Если вы пишете код только на Python, зайдите на стройки по умолчанию для отступов и перенос строк, чтобы значения были совместимы с PEP 8 (<https://www.python.org/dev/peps/pep-0008>). Для этого создайте в домашнем каталоге² 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 позиций
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 нажмите Enter, потом введите `:help` и нажмите Enter для прохождения обучения.

² Для того чтобы зайти домой к каталогу Windows, откройте Vim и введите комманду `:echo $HOME`.

С помощью этих н строек символ переход в новую строку будет добавляться после каждого из 79 символов в строке. Отступы и строены таким образом, что при каждом переходе в табуляции будет добавлено четыре пробела. Если вы находитесь внутри выражения, имеющего отступ, следующая строка будет иметь отступ такого же уровня.

Существует также и дстройки для синтаксиса (известные как python.vim, <http://bit.ly/python-vim>), которые улучшают синтаксис, включенный в Vim 6.1, а также небольшой и дстройка SuperTab (<http://bit.ly/supertab-vim>), позволяющая использовать более удобное вводение кода путем нажатия Tab (или любой другой клавиши, указанной в идентификаторе).

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

Эти идстройки предстаивают основную среду для работы с Python. Если вы имеете копию Vim скомпилированного с параметром +python (или по умолчанию для версии Vim 7 и выше), вы также можете использовать идстройку vim-flake8 (<https://github.com/nvие/vim-flake8>) для выполнения статических проверок кода внутри редактора. Этот идстройк предстаивает функцию Flake8, которая запускается перед запуском Pyflakes (<http://pypi.python.org/pypi/pep8/>) и Pyflakes (<http://pypi.python.org/pypi/pyflakes/>). Идстройк отображает ошибки в нижней части экрана и предложит простой способ переключиться на соответствующую строку.

Если вам удобно, можете звать Vim вызывая 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 истроить его таким образом, чтобы он запускал Pyflakes при записи ошибок и предупреждений в окне quickfix. Рассмотрим пример конфигурации, который позволяет этого достичь, а также видеть 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):

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

- ❑ рефакторинг код и втозолнение с помощью библиотеки горе;
- ❑ быстрое свертывание код Python;
- ❑ поддержка инструмента virtualenv;
- ❑ возможность выполнять поиск в документации Python из пускать код Python;
- ❑ втоматическое 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 махисты 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). Встроена функциональность включает в себя удаленное редактирование (с помощью FTP), календарь, возможность отправлять/читать почту и даже скрипты (нажмите клавишу Esc, затем введите `x` и `doctor`). Среди популярных настроек упомянем плагин YASnippet, необходимый для соотнесения пользовательских синипетов кода и горячих клавиш, и Tramp, предназначенный для работы с удаленными машинами. Его можно расширять с помощью собственного или лекта 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 параметры и конфигурация Python понадобятся. Те, кто еще не работает с Emacs, могут начать свое знакомство с официальным руководством для Emacs (<http://bit.ly/gnu-emacs-tutorial>).

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

- ❑ `python.el` от Фабиана Эзекуеля Гальяны (Fabián Ezequiel Gallina), теперь появляется с Emacs (версия 24.3+), реализует подсветку синтаксиса, отступы, перемещение, взаимодействие с оболочкой и многие другие пространственные особенности Emacs (<https://github.com/fgallina/python.el#introduction>);
- ❑ `Elpy` (<http://elpy.readthedocs.org/>) от Йорген Шефер (Jorgen Schäfer) предоставляет полноценную интерактивную среду для работы с Emacs, который включает в себя инструменты для отладки, проверки кода и втозолнение кода;
- ❑ комплекс файлов исходного кода для Python (<https://www.python.org/downloads/source/>) появляется с пятой версией, которая сполгеряется включением логов

¹ Несомненно Реймонд Хеттингер. Если бы все писали код так, как он рекомендует, мир был бы гораздо лучше.

`Misc/python-mode.el`. Вы можете з грузить его из Интернет к к отдельный ф ёл из приложения launchpad (<https://launchpad.net/python-mode>). Содержит инструменты для р спозн в ния голос , возможность н стройки дополнительных горячих кл виш, т кже позволяет созд ть полноценную IDE для Python (<http://www.emacsWiki.org/emacs/ProgrammingWithPythonModeDotEl>).

TextMate

TextMate (<http://macromates.com/>) — текстовый ред ктор с гр физическим интерфейсом. Созд н н б зе Emacs и р бот ет только для OS X. Имеет удобный пользовательский интерфейс, н йти все его ком нды не сост вит труд .

TextMate н пис н н C++ и был впервые выпущен в 2004 году Алл ном Одг рдом (Allan Oddgard) и Ци р ном Уолшем (Ciarán Walsh). Sublime Text может импортиров ть сниппеты непосредственно в TextMate, Code от комп нии Microsoft — подсветку синт ксис TextMate.

Сниппеты, н пис нные н любом языке, можно собир ть в связ нные группы. Ред ктор можно р сширитъ с помощью сцен риев оболочки: пользователь выделяет текст и отпр вляет его в к честве входного п р метр для сцен рия н ж тием сочет ния кл виш Cmd+| («п йп»). Резуль т р боты сцен рия з менят выделенный текст.

Ред ктор имеет встроенную подсветку синт ксис для Swift и Objective C, т кже (бл год ря Xcode bundle) интерфейс для xcodebuild. Опытный пользователь TextMate не испыт ет з труднений при н пис нии код Python. Новым пользователям, не имеющим дост точного опыта н пис ния код для продуктов комп нии Apple, стоит н ч ть р боту с более новыми кросспл тформенными ред ктор ми, з имствующими многие особенности TextMate.

Atom

Atom (<https://atom.io/>), по мнению его созд телей, — это «уязвимый для х керских т к текстовый ред ктор XXI век ». Выпущен в 2014 году, н пис н н CoffeeScript (JavaScript) и Less (CSS), основ н н Electron (р нее известном к к Atom Shell)¹, который является оболочкой приложения для веб-сервис GitHub, основ нного н io.js и Chromium.

Atom можно р сширитъ с помощью JavaScript и CSS, пользователи могут доб влять сниппеты, н пис нные н любом языке (включ я определения сниппетов в стиле TextMate). Ред ктор хорошо вз имодействует с GitHub. Он пост вляется со встроенной функцион льностью по упрвлению п кет ми и множеством п кетов (2000+). Для р зр ботки н Python рекомендуется использовать Linter

¹ Electron — пл тформ для созд ния кросспл тформенных н стольных приложений с помощью HTML, CSS и JavaScript.

(<https://github.com/AtomLinter/Linter>) вместе с linter-flake8 (<https://github.com/AtomLinter/linter-flake8>). Веб-рэз ботчик может поняться Atom development server (<https://atom.io/packages/atom-development-server>), который в пускет небольшой HTTP-сервер и способен покрыть получившуюся в процессе работы HTML-страницу внутри Atom.

Code

Компания Microsoft выпустила Code в 2015 году. Это бесплатный текстовый редактор с открытым исходным кодом в семействе Visual Studio, основанный на Electron, который можно найти на GitHub. Редактор кроссплатформенный, имеет такие же привязки как и TextMate.

Code построился 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 кроссплатформенный.

IDE

Многие разработчики используют текстовые редакторы, и IDE, переключаясь на IDE для работы на более сложными, крупными или совместными проектами. В таблице 3.2 перечислены основные особенности популярных IDE, в следующих разделах приведены 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), он доступен в бесплатных версиях всех IDE, перечисленных в таблице 3.2. Visual Studio предлагает эту функциональность на всех уровнях¹.

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/Microsoft/PTVS/wiki/Features-Matrix>.

Таблица 3.2. Первый взгляд на IDE

Инструмент	Доступность	Причина использовать
PyCharm/IntelliJ IDEA	<ul style="list-style-type: none"> Открытый API/платная версия для профессионалов. Открытый исходный код/бесплатная версия для сообщества. OS X, Linux, Windows 	<ul style="list-style-type: none"> Программистически идеальное вложение кода. Хорошо поддерживает виртуальные среды. Хорошо поддерживает веб-фреймворков (в платной версии)
Aptana Studio 3 / Eclipse + LiClipse + PyDev	<ul style="list-style-type: none"> Открытый исходный код/бесплатное ПО. OS X, Linux, Windows 	<ul style="list-style-type: none"> Встречается Eclipse. Поддержка Java (LiClipse/Eclipse)
WingIDE	<ul style="list-style-type: none"> Открытый исходный код/бесплатный пробный период. OS X, Linux, Windows 	<ul style="list-style-type: none"> Отличный отладчик (для веб-приложений) — лучший среди перечисленных здесь IDE. Можно расширять с помощью Python
Spyder	<ul style="list-style-type: none"> Открытый исходный код/бесплатное ПО. OS X, Linux, Windows 	<ul style="list-style-type: none"> Аналитик: интегрирован IPython вместе с NumPy, SciPy и matplotlib. IDE по умолчанию в популярных научных дистрибутивах Python: Anaconda, Python(x,y) и WinPython
NINJA-IDE	<ul style="list-style-type: none"> Открытый исходный код/можно вносить пожертвования. OS X, Linux, Windows 	<ul style="list-style-type: none"> Имеет небольшой размер. Сконцентрирован на Python
Komodo IDE	<ul style="list-style-type: none"> Открытый API/текстовый редактор (Komodo Edit) имеет открытый исходный код. OS X, Linux, Windows 	<ul style="list-style-type: none"> Python, PHP, Perl, Ruby, Node.js. Расширения основанные на дополнениях для Mozilla
Eric (the Eric Python IDE)	<ul style="list-style-type: none"> Открытый исходный код/можно вносить пожертвования. OS X, Linux, Windows 	<ul style="list-style-type: none"> Ruby + Python. Небольшой по размеру. Отличный отладчик (научный) — можно выполнять отладку одного потока и выполнять другие
Visual Studio (Community)	<ul style="list-style-type: none"> Открытый API/бесплатная версия для сообщества. Платная версия для профессионалов и предприятий. Только для Windows 	<ul style="list-style-type: none"> Отличная интеграция с инструментами языка программирования Microsoft. Функциональное вложение кода посредством IntelliSense.

Инструмент	Доступность	Причина использовать
		<ul style="list-style-type: none"> Управление проектами и поддержка приложений звёртыванием, включая инструменты для планирования спринтов и шаблоны мифетов в версии Enterprise. <p>Примечание. Имеется недостаток: нельзя использовать виртуальные среды во всех версиях, кроме Enterprise (с моей дорожной)</p>

PyCharm/IntelliJ IDEA

PyCharm (<http://www.jetbrains.com/pycharm/>) — наш любимая IDE для Python. В качестве основных причин использовать именно ее можно привести практические инструменты визуализации кода, а также качество инструментов для веб-разработки. Установки и учного сообщества рекомендуют бесплатную версию (которая не имеет инструментов для веб-разработки), поскольку она вполне подходит, однажды они выбирают Spyder.

PyCharm является компонентом JetBrains, также известной как IntelliJ IDEA. Представляет собой proprietарную IDE для Java, которая конкурирует с Eclipse. PyCharm (выпущен в 2010 году) и IntelliJ IDEA (выпущен в 2001-м) имеют общую базу кода, и большую часть функциональности PyCharm можно использовать в IntelliJ IDEA бесплатно и без установки Python (<http://bit.ly/intellij-python>).

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

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

Aptana Studio 3/Eclipse + LiClipse + PyDev

Eclipse написан на Java, выпущен в 2001 году компанией IBM как открытая и гибкая IDE для Java. PyDev (<http://pydev.org/>), аддукция Eclipse для разработки на Python, выпущен в 2003-м Алексом Тотиком (Aleks Totic), который впоследствии передал её Фабио Задрожному (Fabio Zadrozny). Это наиболее популярная аддукция Eclipse для разработки на Python.

Несмотря на то что сообщество Eclipse не перечит, когда кто-то голосует за использование IntelliJ IDEA на форумах, где спорят о том, что лучше эти две IDE, Eclipse все еще считается наименее распространенной IDE для Java. Этоично для разработчиков Python, в отличие от существующих инструментов для Java, поскольку многие популярные инструменты (например, Hadoop, Spark и их производные версии) построены с инструкциями и конструкциями для разработки с помощью Eclipse.

В Studio 3 от Aptana (<http://www.aptana.com/products/studio3.html>) встроен версия PyDev, предлагающая собой набор инструментов с открытым исходным кодом. Он построился с Eclipse (предоставляет IDE для Python (и Django), Ruby (и Rails), HTML, CSS и PHP). Основные преимущества включают в себя Aptana, Appcelerator, — Appcelerator Studio, проприетарная мобильная платформа для HTML, CSS и JavaScript, требующая покупки месячной лицензии (но только вы выпустите свое приложение). Они поддерживают и PyDev с Python, но это не является приоритетным. Если вам интересуется Eclipse и вы в основном работаете с JavaScript, создавайте приложения для мобильных платформ и иногда используйте Python (особенно если работаете с Appcelerator), Studio 3 от Aptana отлично подойдет.

LiClipse появился на свет более года назад и предоставляет более качественную поддержку нескольких языков в Eclipse, а также упрощает доступ к полностью черным темам (то есть в дополнение к фону для текста меню и группаницы также будут черными). Этот проприетарный набор инструментов для Eclipse не имеет 3-дружным; часть стоимости лицензии (опционально) идет на то, чтобы PyDev осталась полностью бесплатным и имел открытый исходный код (лицензия EPL, как и у Eclipse). Построился вместе с PyDev, поэтому пользователям Python не нужно устанавливать его самостоятельно.

WingIDE

WingIDE (<http://wingware.com/>) — это IDE для Python. Вполне возможно, что это второй 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 использования WingIDE называются отладчик и небольшой объем этой IDE.

Сред Wing выпущен в 2000 году компанией Wingware, не имеющей Python, C и C++. Поддерживает расширения, но пока еще не имеет репозитория инструментов, поэтому ее пользователям придется искать существующие пакеты в блогах, а также в учетных записях GitHub.

Spyder

Spyder (<https://github.com/spyder-ide/spyder>) (разработанная к Scientific PYthon Development EnviRonment — научной средой разработки для Python) — это IDE, предложенная для работы с научными библиотеками Python. Написана на Python Кристофером Сирдобой (Carlos Cyrdoba), имеет открытый исходный код (лицензия MIT) и предлагает такие возможности, как встроение кода, подсветка синтаксиса, обзор методов классов и функций, исследование объектов. Доступ к другой функциональности можно получить с помощью инструментов 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 Linux — Anaconda, Python(x,y) и WinPython.

NINJA-IDE

NINJA-IDE (<http://www.ninja-ide.org/>) (название представляет собой рекурсивный кроним 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 разработан с использованием Python и Qt, имеет открытый исходный код (лицензия GPLv3) и специальную легковесную версию без инструментов. Одной из особенностей 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 инструментов (пользователи могут добавлять только необходимые инструменты).

Разработка находится в стадии разработки, но новую версию NINJA-IDE v3 планируется выпустить в 2016 году и в данный момент все еще идут активные переговоры на счет listserv для NINJA-IDE (<http://bit.ly/ninja-ide-listserv>). Для многих членов сообщества, включая коммандированных 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 лицензию с открытым исходным кодом (под общественной лицензией Mozilla).

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

Eric (the Eric Python IDE)

Eric – это IDE с открытым исходным кодом (лицензия GPLv3), который активно развивается уже более десяти лет. Написан на Python и базе набора инструментов Qt GUI, в который интегрирован редактор Scintilla. Назван в честь Эрика Айдла (Eric Idle), член группы «Монти Пайтон» (Monty Python), также в знакуважения к IDLE IDE, поставляемой с дистрибутивами Python.

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

Как NINJA-IDE и Komodo IDE, среди Eric специальность 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 использовать Matplotlib для интерактивного создания графиков в этой IDE, используйте backend Qt4:

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

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

Ссылка на официальную документацию 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 гидродинамическими моделями).

Visual Studio

Профессиональные программы компании, работающие с продуктами компании Microsoft в операционной системе Windows, зайдут воспользоваться Visual Studio (<https://www.visualstudio.com/products>). Он написан 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 организаций.

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

Убедитесь, что установлены Python Tools for Visual Studio (PTVS) (<https://www.visualstudio.com/en-us/features/python-vs.aspx>) (этот пункт не указан по умолчанию в списке пользовательских инструментов 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 он не был включен в предыдущий раздел, поскольку многие считают ее достаточноенной для того, чтобы использовать для производственных проектов, как другие IDE. Однако он отлично подходит для обучения. IPython по умолчанию встроен в Spyder (может быть встроен и в другие IDE). Они не заменяют интерпретатор Python, скорее подгружают выбранные пользователем оболочки интерпретатора с помощью дополнительных инструментов и функциональности.

IDLE

IDLE (Integrated Development and Learning Environment — интегрированная среда для разработки и обучения; имеет также ссылку на автора Эрика Айдла (Eric Idle)) (<http://docs.python.org/library/idle.html#idle>) — часть стандартной библиотеки Python. Поставляется вместе с Python.

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

Предоставляет следующие возможности:

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

IPython

IPython (<http://ipython.org/>) – полезный набор инструментов, который поможет вам коснуться активной среды Python.

Перечислим его основные компоненты:

- ❑ мощные оболочки Python (небез консоли и Qt);
- ❑ блокнот небез Интернета, имеющий такую же основную функциональность, как и терминлья оболочки, также поддерживая мультимедиа, текст, код, тематические выделения и встроенные графики;
- ❑ поддержка интерактивной визуализации данных (при соответствующем конфигурировании в шиферах, созданные в Matplotlib, будут появляться в окнах) и применения инструментов графического пользователя в графическом интерфейсе;
- ❑ гибкие встроенные интерпретаторы, предназначенные для загрузки в широких проектах;
- ❑ инструменты для выполнения высокогородневых и интерактивных параллельных вычислений.

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

```
$ pip install ipython
```

bpython

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

- ❑ встроенный подсветка синтаксиса;
- ❑ автоматическое создание отступов и втоэолнение кода;
- ❑ ожидающий список правил 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 терминалной 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 предоствляют 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 в глобальный каталог `site-packages` (место, где хранятся установленные пользователем пакеты Python) остается упорядоченным и вы сможете работать над проектом, который требует наличия фреймворка Django 1.3, в то же время поддерживая проект, требующий Django 1.0.

Команды `virtualenv` позволяют достичь этого, создавая отдельный каталог, содержащий гибкую ссылку на исполняемый файл Python, копию рабочего места для библиотек Python (добавляет его в начальную переменной среды `PATH` при активизации, в темновую среду в исходное состояние при деактивации). Вы также можете использовать установку новых зависимостей 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 активации виртуальных сред Python отличается в разных операционных системах.

Mac OS X и Linux. Вы можете указать версию Python с помощью аргумента `--python`. Далее используйте следующий скриптик, 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 скриптов, созданных в локальной среде¹.

Запустите PowerShell от имени администратора и введите следующий код:

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

¹ Если в дальнейшем будет возникнуть необходимость использовать 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

Когда только вы активизировали виртуальную среду, первым исполняемым файлом будет тот, который расположен в только что созданном к этому логе *my-venv*. Этот файл устанавливает библиотеки в следующую папку:

- ❑ *my-venv/lib/python3.4/site-packages/* (в системах POSIX¹);
- ❑ *my-venv\Lib\site-packages* (в Windows).

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

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

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

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

Команда `pip` устанавливает перечисленные зависимости, переопределяя зависимости, указанные в подключенном файле, в том случае, если возникли конфликты. Зависимости, указанные в файле *requirements.txt*, предписаны для установки всей среды Python. Для того чтобы установить зависимость 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 (портативный интерфейс операционных систем). Включает набор стандартов IEEE, указывающих на поведение и интерфейсы для простых команд оболочки, ввод/вывод, работы с потоками и для других сервисов и утилит. Большинство дистрибутивов Linux и Unix совместимы с POSIX, Darwin (операционные системы, лежащие в основе Mac OS X и iOS), был совместим с версией 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 установке для Python (<https://packaging.python.org/>). Команду ruenv, который распространяется вместе с Python, который не изменяет virtualenv (функционально является зависимостью для virtualenv), поэтому эти инструкции работают для всех версий Python.

ruenv

Инструмент ruenv (<https://github.com/yuuu/ruenv>) позволяет работать с несколькими версиями интерпретаторов Python одновременно. Это решает проблему, возникающую при наличии нескольких проектов, когда каждый требует разных версий Python, но в них все еще придется использовать виртуальные среды в том случае, если в библиотеках возникнет конфликт из 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 ruenv не ограничен только версиями CPython, он также устанавливает интерпретаторы PyPy, Anaconda, Miniconda, Stackless, Jython и IronPython.

Работа ruenv заключается в том, что он заполняет контекст лога shims вспомогательной версией интерпретатора Python и исполняемыми файлами, вроде pip и 2to3. Эти файлы можно найти, если контекст лога находится в переменной среды \$PATH. Вспомогательная функция — это проходящая функция, которая интерпретирует текущую ситуацию и выбирает самую подходящую функцию для выполнения желаемой задачи. Например, когда система ищет программу с именем python, она сначалаглядит внутрь контекста лога shims и использует вспомогательную версию, которая в свою очередь передает команду ruenv. После этого ruenv 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 идентификатор ruenv-virtualenv (<https://github.com/yuuu/ruenv-virtualenv>): в том числе позволяет создавать различные среды, которые позволяют использовать существующие инструменты ruenv для переключения между ними.

Autoenv

Autoenv (<https://github.com/kennethreitz/autoenv>) позволяет легко управлять различными настройками среды из пределов определенной видимости virtualenv. Переопределяет команду оболочки cd таким образом, что, когда вы переходите в контекст лога, содержащий

файл с расширением .env (например, устанавливая значение переменной среды PATH с помощью URL для ботов других), Autoenv в том числе активизирует среду. Когда вы выходите из контекст лога, вызвав в 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
```

Затем открыть новое окно консоли.

virtualenvwrapper

Инструмент virtualenvwrapper (<http://bit.ly/virtualenvwrapper-docs>) предлагает набор команд, который расширяет виртуальные среды Python, чтобы им было легче управлять. Он помогает все виртуальные среды в один контекст лога и предоставляет пустые функции перехват (их можно запустить до или после создания/активации виртуальной среды или проекта, например функция перехват может установить новые переменные среды путем поиск файла с расширением .env внутри контекст лога).

Проблема с размещением функций с установленными объектами ключается в том, что пользователь должен каким-то образом получить доступ к другим сценариям, чтобы полностью скопировать среду на другую машину. Это может пригодиться для общего сервера, если все среды помещены в единый контекст лога 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 в 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 ее на другую 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). Немного других полезных функций из документации можно найти 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 базы данных или серверы). 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 включают пользовательские рецепты в исходный код), а также конфигурационный файл `buildout.cfg` и скрипты `bootstrap.py` в каталоге верхнего уровня. Если вы запустите скрипты, введя команду `python bootstrap.py`, он прочтет конфигурационный файл, чтобы определить, какие рецепты нужно использовать, а также настройки конфигурации для каждого рецепта (например, определенные файлы компилятора и файлы для связывания библиотек).

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

egg¹, что можно пропустить в новых версиях Python, которые используют рхивы wheels. Обр титесь к руководству Buildout (<http://www.buildout.org/en/latest/docs/tutorial.html>) для получения более подробной информ ции.

Conda

Инструмент Conda (<http://conda.pydata.org/docs/>) похож на pip, virtualenv и Buildout одновременно. Пост вляется 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 зарезных репозиториях (pip получает их из <http://pypi.python.org>, conda из <https://repo.continuum.io/>), имеют зарезные формы, поэтому эти инструменты неизвестны нам.



В таблице по адресу <http://bit.ly/conda-pip-virtualenv>, 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 файл конфигурации conda-build называется рецептом (не ограничен использованием только лишь инструментов Python). В отличие от Buildout, код записан в специальной оболочки (это не код Python). Конфигурация приводится в формате YAML² (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 с библиотекой по умолчанию), предлагающей полезный интерфейс для distutils (<https://docs.python.org/3/library/distutils.html>) instead of 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>) называется YAML Ain't Markup Language — «не просто еще один язык разметки».

Основное преимущество conda перед pip и virtualenv оценят пользователи Windows — библиотеки Python, созданные для распространения в C, могут быть предствлены 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, в этом — пакеты, которые размещаются в PyPI.

Docker

Инструмент Docker (<https://www.docker.com/>) помогает изолировать среду (как virtualenv, conda или Buildout), но вместо того, чтобы предоставить виртуальную среду, предлагает контейнер Docker. Контейнеры проще изолировать, чем среды. Например, вы можете запустить несколько контейнеров — и у каждого будет свой сетевой интерфейс, привилегии будут иметься 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 запустите контейнеры Docker в OS X, Windows или на удаленном хосте, можно добиться Docker Machine (<https://docs.docker.com/machine/>) (позволяет взаимодействовать с виртуальными машинами¹, запущенными в Docker Engine).

Контейнеры Docker изначально были основой контейнеров Linux Containers, которые были связаны с помощью оболочки chroot (<https://en.wikipedia.org/wiki/Chroot>).

chroot — это подобие команды virtualenv на системном уровне: позволяет сделать так, чтобы корневой каталог (/) расположился по адресу, указанному пользователем, 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 (и логичен PyPI).

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

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

если вы хотите построить и протестировать свой проект для нескольких версий зависимостей, можете создать основной образ Docker с виртуальной средой¹ (средой Buildout или conda), содержащей все остальные зависимости.

Вы увидите следующее от этого образа: все остальные образы, добавленные в последнем уровне, одну изменяющуюся зависимость. В результате все увиденные контейнеры будут содержать только отличающиеся библиотеки, разделая при этом содержимое основного образа. Для получения более подробной информации об этом обратитесь к документации Docker по адресу <https://docs.docker.com/>.

¹ Виртуальная среда внутри контейнера Docker изолирует вашу среду Python, сохранив версию Python операционной системы, которой может понадобиться для вших приложений. Поэтому прислушайтесь к нашему совету ничего не устанавливать с помощью pip (или чего-то еще) в системный каталог лог 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 на стоящем 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 binary файлы в PyPI, рассмотрим различные 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 приведем хороший тон, связанные с журнализмом, перечислим основные различия между доступными лицензиями для открытого исходного кода. Это поможет вам писать код, который впоследствии можно будет легко использовать и распространять.

Стиль кода

Питонисты (вертены разработки на Python) рады тому, что их язык не столько понятен, — люди, которые никогда не знали языка Python, 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 позже) и питонских идиомах. Если питонист утверждает например, что он не питонский, это обычно означает, что строки не соответствуют спрощенненным принципам и не являются читаемыми. Конечно, «глупая последовательность — пугало маленьких умов»¹. Недородное следование PEP может снизить читаемость и понятность.

¹ Цитату изначально приведена Ральфом Уолдо Эмерсоном (Ralph Waldo Emerson) в эссе *Self-Reliance*. Присутствует в PEP 8 для того, чтобы подтвердить, что здравый смысл вживе руководствует 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, структуре кода, пустые обработчики (табуляция против пробелов) и другие логичные темы.

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

Написать код с учетом принципов PEP 8 — хорошая идея (помогает зарубежным ботчикам создавать более стильный 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 терминалlettную комманду:

```
$ pip3 install pep8
```

Посмотрим 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 даны 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 комманду auto_pep8, которая в том числе переформатирует код согласно PEP 8. Установить ее можно так:

```
$ pip3 install autopep8
```

Чтобы переформатировать файл (перезапись в оригинале), введите следующую комманду:

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

Если вы не добавите флаг `--in-place`, это будет считываться программой выводить модифицированный код в консоль (или записывать в другой файл). Флаг `--aggressive` выполнит

более существенные изменения, его можно применить несколько раз для получения зн чительного 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 форизмов, а не 20 (последний не был записан).

Редкая история «Дзен Питона» увековечен в статье Барри Уорсона (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 за пут иное.

Одноуровневое лучше, чем вложенное.

Резкое лучше, чем плотное.

Читаемость имеет значение.

Особые случаи не должны быть предпочтительнее общих.

При этом конкретность важнее безупречности.

Ошибки никогда не должны оставаться незамеченными.

Если не замечиваются явно.

Встретив двусмысленность, отбрось искушение угадывать.

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

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

Сейчас лучше, чем никогда.

Хотя никогда не лучше, чем прямо сейчас.

Если результатию сложно объяснить — идея плоха.

Если результатию легко объяснить — идея, возможно, хороша.

Пространство имён — отличная штука! Будем делать их побольше!

Для того чтобы увидеть пример использования каждого из этих форизмов, обратитесь к презентации Хантера Бланкс (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 «Дзен Питона», другие основаны на здравом смысле. Они подтверждают наш принцип работы: при написании кода Python выбирайте наиболее очевидный способ его представления из имеющихся вариантов.

Явное лучше, чем неявное

В Python предпочтителен и более явный способ выражения:

Плохой код	Хороший код
<pre>def make_dict(*args): x, y = args return dict(**locals())</pre>	<pre>def make_dict(x, y): return {'x': x, 'y': y}</pre>

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 быстрокции списков (иногда называемых списковыми включениями — list comprehensions)) позволяет и даже поощряется за их красоту и изящность, но признаком хорошего тона будет размещение отдельных выражений на разных строках. Это поможет создать более простые для понимания результаты¹, когда подобное выражение изменяется:

Плохой код	Хороший код
<code>print('one'); print('two')</code>	<code>print('one') print('two')</code>

Продолжение ↗

¹ Результат — это утилиты оболочки, которые сравнивают два файла и показывают отличияющиеся строки.

(Продолжение)

Плохой код	Хороший код
<code>if x == 1: print('one')</code>	<code>if x == 1: print('one')</code>
<code>if (<complex comparison> and <other complex comparison>): # сделать что-нибудь</code>	<code>cond1 = <complex comparison> cond2 = <other complex comparison> if cond1 and cond2: # сделать что-нибудь</code>

Повышение читаемости кода среди питонистов ценится гораздо выше, чем увеличение объема и несколько быт (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 практики HowDoI (более подробно описаны здесь в разделе «HowDoI» в главе 5) Бен Глейтмана (Ben Gleitzman) показывает, когда замечательные 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 мной стоял вопрос, который предоставляет сценарий командной строки, позволяющий найти в Интернете (по умолчанию с помощью Stack Overflow) способ выполнить задачу по программированию. Функция `format_output()` подсвечивает синтаксис, просматривая теги вопроса и предмет строки, которую смог разобрать

лекссер (также он называется *токенайзером*; теги `python`, `java` или `bash` позволяют определить лекссер, который нужно использовать для разбиения и подсвечивания кода), в то время, если он достигнет сбоя, пробует определить язык по своему коду. Когда прогресс достигает оператора `try`, он может пойти по одному из трех путей:

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

Часто форизм «не замечавшиеся» препятствует чрезмерному выявлению ошибок. Рассмотрим пример (можете попробовать запустить его в *отдельном окне консоли* — там будет проще прервать выполнение, когда вы во всем увидите):

```
>>> 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 игнорирует их. Соответственно, он проигнорирует множество в таких попытках прервать его работу. Это не просто проблема с прерываниями — блок `except` также может скрывать ошибки, что вызовет проблемы в будущем (их старт не так трудно диагностировать). Поэтому *не замалчивайте ошибки*: всегда явно указывайте имена исключений, которые хотите поймать, и обрабатывайте только их. Если вы хотите просто запустить в журнал или кого-то еще убедиться в наличии исключения и вызвать его повторно, как в следующем снippetе, тогда все в порядке. Только не замечавшие ошибки (не обрабатываться ими 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 различными способами:

1**2****3****4**

```
def func(positional, keyword=value, *args, **kwargs):  
    pass
```

- 1** *Позиционные аргументы* обязательны и не имеют значений по умолчанию.
- 2** *Аргументы с ключевым словом* необязательны и имеют значения по умолчанию.
- 3** *Список с произвольным количеством аргументов* необязательны и не имеет значений по умолчанию.
- 4** *Словарь с произвольным количеством аргументов* с ключевым словом необязательен 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 произвольный порядок. Например, пользователь без труда вспомнит, что у функций `send(message, recipient)` или `point(x, y)` должны быть два аргумента, т.к. же порядок этих аргументов.

Антишблон: 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.

Антишблон: можно отправлять аргументы в произвольном порядке, но не указывать их имена явно, например `send("42", "Frankie", "Benji", "Trillian")`, переслав скрытую копию пользователю с именем Триллиан. Можно передать именованные аргументы в непривильном порядке, например `send("42", "Frankie", bcc="Trillian", cc="Benji")`. Если у вас нет веской причины делать это, лучше всего использовать вariant приближенный к определению функции: `send("42", "Frankie", cc="Benji", 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`, который укажет 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", "Benji", "Trillian")`. В теле функции конструкция `args` будет равна списку (`"Frankie", "Benji", "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 ее к `send("42", ["Benji", "Frankie", "Trillian"])`.

- ❑ *Словарь с произвольным количеством аргументов с ключевым словом.* Такой словарь определяется с помощью конструкции `**kwargs`, который укажет 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 журнализации. Средств формирования на разных уровнях могут принять необходимую 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 считаются позиционными, какие — необязательными, зависит только от программиста, который пишет функцию. От него зависит различие 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 поистине является с богатым набором инструментов (зачем его любят хакеры), который позволяет в нем 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 получите в результате, должно быть более ясным, 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, когда применять эти знания *нельзя*.

К примеру, 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`. Этой философии сильно отличается от той, что присущ высокозращенным языкам вроде Java, — они имеют множество механизмов, предотвращающих неверное использование. Философия Python сосредоточена в фразе «Мы все — ответственные пользователи».

Это не значит, что ни одно свойство не считается заскрытым и что в Python нельзя перезаписать инкапсуляцию. Наоборот, вместо того чтобы возводить бетонные стены между своим и чужим кодом, сообщество Python предпочитает использовать гораздо более гибкие соглашения, которые указывают, каким образом элементам нельзя получить доступ непосредственно.

Основным соглашением для закрытых свойств и лейкейрэлизации является добавление к именам всех подобных элементов нижнего подчеркивания (например, `sys._getframe`). Если клиентский кодрушит это правило и получит 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 областей и упрощает внесение изменений в код. Всегда можно сделать из закрытого свойства открытым, обратное же действие выполнить гораздо сложнее.

Возвращайте значения из одной точки

Когда сложность функции увеличивается, в которой вы можете встретить несколько возврата `return` в теле этой функции. Однако для того, чтобы ее было проще понять и прочесть, возвращайте осмысленные значения из минимум возможно количества точек.

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

Однако везде, где это возможно, имейте только одну точку выхода — сложно выполнить отладку для функций, когда в них член нужно определить, к какому возврату `return` ответственно за результат. И наличие единой точки выхода из функции также поможет избавиться от некоторых ветвей кода, поскольку наличие промежуточных точек выхода, возможно, не мешает ни то, что необходимо провести подобный рефакторинг. Код в следующем примере нельзя назвать плохим, но его можно сделать более чистым (как это показано в комментах):

```
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 просты иены, 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. страницу

«Проверка значений на правдивость» (<http://docs.python.org/library/stdtypes.html#truth-value-testing>) — там представлен список значений, которые при сравнении возвращаются к `False`.

Плохой код	Хороший код
<pre>if attr == True: print 'True!'</pre>	<pre># 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'</pre>
<pre>if attr == None: print 'attr is None!'</pre>	<pre># или явно проверяем на значение None if attr is None: print 'attr is None!'</pre>

Получаем доступ к элементам массива

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

Плохой код	Хороший код
<pre>>>> d = {'hello': 'world'} >>> >>> if d.has_key('hello'): ... print(d['hello']) # prints 'world' ... else: ... print('default_ value') ... world</pre>	<pre>>>> 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</pre>

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
<pre># Отфильтруем все элементы, # чье значение превышает 4 a = [3, 4, 5] b = [] for i in a: if i > 4: b.append(i)</pre>	<pre># 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)</pre>
<pre># Добавим 3 к каждому элементу списка a = [3, 4, 5] for i in range(len(a)): a[i] += 3</pre>	<pre># Здесь также прозрачнее a = [3, 4, 5] a = [i + 3 for i in a] # Или: a = map(lambda i: i + 3, a)</pre>

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 значения¹, 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 окаться 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 не закрытую 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, в этом случае это значение зависит от того, что говорит шинильник X! Честно говоря, любой, кто использовал консоль для отладки кода в полевых условиях, быстро оценит ограничение в 80 символов (при котором строки в консоли не переносится) и на деле будет использовать 75–77, чтобы можно было увидеть номерацию строк в Vim.

Плохой код	Хороший код
<pre>french_insult = \ "Your mother was a hamster, and \ your father smelt of elderberries!"</pre>	<pre>french_insult = ("Your mother was a hamster, and " "your father smelt of elderberries!")</pre>
<pre>from some.deep.module.in.a.module \ import a_nice_function, \ another_nice_function, \ yet_another_nice_function</pre>	<pre>from some.deep.module.in.a.module import (a_nice_function, another_nice_function, yet_another_nice_function)</pre>

Одн ко з ч стую необходимость разбивая длинные логические строки удалять то, что вы пытаетесь выполнить слишком много действий разом, 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 показаться более интересным для новичков (если только они не голландцы¹). Поэтому в нем необходимо освоить хорошие идиомы.

Распаковка

Если вы знаете длину списка или кортежа, можете присвоить именем элементы 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), левая сторона может содержать любое количество 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 разделять 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 «Дзен Питон». Гвидо, напоминая BDFL, — голландец.

В Python 3 в PEP 3132 (<https://www.python.org/dev/peps/pep-3132/>) был предложен новый метод расширенной проверки:

```
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 в м необходимо присвоить к кое-то зн чение во время р сп ковки, но с м переменн я не нужн , воспользуйтесь двойным подчеркив нием (_):

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



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 `list *` для того, чтобы создать список, состоящий из одинаковых неизменяемых элементов:

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

Однократные объекты должны иметь одинаковые значения хэш . 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 в чем-то устроит. Поэтому используйте списковое включение:

Плохой код	Хороший код
<pre>>>> four_lists = [[]] * 4 >>> four_lists[0].append("Ni") >>> print(four_lists) [['Ni'], ['Ni'], ['Ni'], ['Ni']]</pre>	<pre>>>> four_lists = [[] for _ in range(4)] >>> four_lists[0].append("Ni") >>> print(four_lists) [['Ni'], [], [], []]</pre>

Распространенный идиом для создания строк состоит в том, чтобы использовать функцию `str.join()` для пустой строки. Для этого идиом может быть применен 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 булевые проверки на наличие в списке и множестве выглядят идентично, `foo` в `y` учитывает тот факт, что множества (и словари) в Python являются хеш-таблицами¹, производительность для этих двух примеров будет различной. Python должен пройти по каждому элементу списка в поисках совпадения, и это занимает много времени (это заметно при увеличении размера коллекций). Но поиск ключей во множестве может быть выполнен быстро с помощью поиска по хешу. Кроме того, множества и словари не могут содержать повторяющихся значений и идентичных ключей. Для получения более подробной информации поинтересуйтесь семинаром на эту тему на ресурсе Stack Overflow (<http://stackoverflow.com/questions/513882>).

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/>) предложен оператор `with` и протокол управления контекстом (в версиях 2.5 и выше) — идиом, позволяющая менять блоки `try/finally` на более читаемый код. Протокол состоит из двух методов, `__enter__()`

¹ Кстати, именно поэтому только хешируемые объекты можно хранить во множествах или использовать как ключи для словарей. Чтобы внести объекты Python в словарь хешируемыми, определите функцию-член объекта `__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, связывая вызов метода `close()`, подавляя исключения (в Python 3.4 и выше) и перенаправляя 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 файлов¹, мы можем использовать это выражение непосредственно, не закрывая файл с mostостоятельно:

¹ В этом случае метод `__exit__()` вызовет метод обертки для ввод/вывод `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 недежный язык. Однако некоторые ситуации могут быть непонятны для новичков: кто-то из них создаст ошибку, 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 причине).

Изменяемые аргументы по умолчанию

Ниболее чистый сюрприз, с которым сталкиваются новые программы Python, — это отношение Python к изменяемым аргументам по умолчанию в определениях функций.

Что вы написали:

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

Чего вы ожидали:

```
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, не при каждом ее вызове (когда 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 в этом нужно сделать вместо этого? Создайте 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 неподозрительно действовать (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 окружющей глобальной области видимости).

Что вы написали:

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

Чего вы ожидали:

```
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 ...

82 Часть II. Переходим к делу

Что происходит на самом деле:

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

Создаются пять функций, все они умножают `x` на 4. Почему? В Python 3 мыкания имеют *позднее связывание*. Это говорит о том, что значения переменных, использованных в 3 мыканиях, определяются в момент вызовов внутренней функции.

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

Особенно неудобно то, что в нем может показаться, будто ошибки к какому-то связанным лямбда-выражениями (<https://docs.python.org/3/tutorial/controlflow.html#lambda-expressions>). Функции, созданные с помощью лямбда-выражений, не отличаются от других. Фактически то же самое поведение проявляется и при использовании синтаксиса обычного `def`:

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

Что в нем нужно сделать вместо этого? Наиболее общее решение, возможно, нет «костылем» — временным впринципе устраниния проблемы. Из-за уже упомянутого поведения Python, связанныго с определением аргументов по умолчанию для функций (см. предыдущий пункт «Изменяемые аргументы функций»), вы можете создать 3 мыкание, 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 3 мыкания вели себя подобным образом. Позднее связывание может быть полезным во многих ситуациях (например, в проекте Diamond, см. пункт «Пример использования 3 мыкания (когда подводный камень вовсе не подводный камень)» на с. 136). Наличие уникальных функций в цикле, к сожалению, может привести к сбоям.

Структурируем проект

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

По какому принципу функции должны размещаться в модулях? Куда новые перемещаются по проекту? Какие функции могут быть сгруппированы и изолированы? Отвечая на эти вопросы, вы можете создать проект, который будет выглядеть в конечный продукт.

В книге *Python Cookbook* есть глава, посвященная модулям и проектам (<http://bit.ly/python-cookbook-ch10>), в которой подробно описываются, как работают выражения `__import__` и упаковка. Цель этого раздела — осветить основные спектры системы модулей и импортирования Python, необходимые для структурирования в них проектов. Далее мы рассмотрим различные подходы к сборке кода, который легко будет распространяться и тестируется.

Благодаря тому, как в Pythonжен 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 практическая задача — создать различные стили проекта и продумать их взаимодействие.

Модули

Модуль — это один из основных уровней бестратности в Python. Уровни бестратности позволяют программисту разбивать код на части, которые содержат связанные между собой функции и классы.

Например, если один уровень проекта предназначен для взаимодействия с пользователями, другой обрабатывает более высоком уровне, а более логичным способом разделения этих двух слоев является размещение всей функциональности, связанной со взаимодействием, в одном файле, всех низкоуровневых операций — в другом. Такая группировка заметит их в двух разных модулях. Файл для взаимодействия с тем импортирует файл для низкоуровневой обработки с помощью выражения `import module` или `from module import attribute`.

К сожалению, вы пустите в ход выражение `import`, вы нечнете пользоваться модулями. Модули могут быть либо встроенными (вроде `os` и `sys`), либо сторонними (вроде `Requests` или `NumPy`), либо внутренними модулями проекта.

Далее покажем 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 код (<https://www.python.org/dev/peps/pep-0008/>) присвивте модулям короткие имена, которые не начинаются со строчной буквы. И убедитесь, что не используете специальные символы вроде точки(.) или вопросительного знака(?), поскольку это может нарушить вид Python для модулей. Поэтому в нем следует избегать имен файлов вроде `my.spam.py`¹ (Python попытается найти файл `spam.py` в текущем каталоге с именем `my`, это неверно). В документации Python (<http://docs.python.org/tutorial/modules.html#packages>) более подробно описано это.

Импортирование модулей. Помимо следов ния некоторым ограничениям в именовании, для использования файла Python в качестве модуля не требуется больше ничего особенного. Однако понимать механизм импортирования будет нелегким. Во-первых, выражение `import modу` не ищет определение `modу` в файле с именем `modу.py` в том же каталоге, где находится и вызывающая сторона, если такой файл существует. При неудаче интерпретатор Python будет рекурсивно искать файл `modу.py` в пути поиска Python (<https://docs.python.org/2/library/sys.html#sys.path>) и генерирует 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.
```

К сожалению, файл `modу.py` будет найден, интерпретатор Python запустит модуль в ограниченной области видимости. Любое выражение верхнего уровня в файле `modу.py` будет выполнено, включая другие выражения импорта, если только они существуют. Определения функций и классов находятся в словаре модуля. На конец, переменные функции и классы модуля будут доступны вызывающей стороне с помощью *пространства имен* модуля — основной концепции программирования, которая особенно эффективна в Python. Пространство имен предоставляет облегченный

¹ Если хотите, можете называть файл `my_spam.py`, но даже не таким другим — нижним подчеркиванием — не следует злоупотреблять в именах модулей (нижнее подчеркивание не вводит мысль, что перед ми имя переменной).

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

Во многих языках директивы ставят препроцессор, по сути, скопировав содержимое включенного файла в код вызывающей стороны. В Python все происходит иначе: включенный код изолируется в пространстве имен модуля. Результатом выполнения выражения `import modu` не будет объект модуля с именем `modu`, который будет находиться в глобальном пространстве имен, его трибуты будут доступны с помощью точечной записи. Например `modu.sqrt` — это объект `sqrt`, определенный внутри файла `modu.py`. Это означает, что в module, к которому привило, не нужно волноваться о том, что включенный 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 пространство имен:

- `dir(object)` возвращает список трибутов, к которым объект может получить доступ;
- `globals()` возвращает словарь трибутов, находящихся в данный момент в глобальном пространстве имен, также их значения;
- `locals()` возвращает словарь трибутов в текущем локальном пространстве имен (например, внутри функции), 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, к сожалению, считается признаком плохого тона: наличие конструкции `import *` усложняет чтение кода, делает ее зависимостью более связанными и может затруднить (перезапись) существующие определенные объекты новыми описаниями из импортированного модуля.

Нотация `from modu import func` — это способ импортировать только необходимые в module трибуты в глобальное пространство имен. Он гораздо безопаснее нотации `from modu import *`, поскольку указывает, что именно импортируется в глобальное пространство имен. Единственное ее преимущество перед более простой нотацией `import modu` в том, что он экономит вам немногого времени.

В таблице 4.1 сравниваются различные 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

К упоминается в разделе «Стиль код» в начале этой главы, читаемость — один из основных особенностей Python. Читаемый код не содержит бесполезного текста. Но не следует максимизировать его сокращение 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 придется, следуйте избегая некоторых ошибок.

- *Большое количество запутанных циклических зависимостей.* Если для в ших 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 в свою очередь зависит от файла `furn.py`. В таком случае вам нужно использовать выражение `import` внутри методов, чтобы 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 скользящих изменений для того, чтобы к ним адаптироваться, и очевидно, что в своем коде класс `Carpenter` вы делаете слишком много предположений о классе `Table`.
- *Избыточное использование глобального состояния или контекста.* Вместо явной передачи различных (высота, ширина, тип, древесина) друг другу классы `Table` и `Carpenter` полагаются на глобальные переменные, которые модифицируются на лету различными генераторами. Вам придется перебрать все объекты, имеющие доступ к этим глобальным переменным, чтобы понять, почему прямоугольный стол стоит в другом месте, и обнаружить, что это сделал код, который отвечает за работу блонов.
- *Спагетти-код.* Вложенные условия `if`, расположенные на нескольких строках подряд, и циклы `for`, содержащие большое количество скопированных

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

- *Равиoli-код*. Такой код в Python встретить более вероятно, чем спагетти-код. *Равиoli-код* состоит из сотен небольших логических фрагментов, связанных между собой 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 *равиoli-кодом*.

Упаковка

Python предоставляет довольно понятную систему упаковки, которая расширяет механизм модулей таким образом, что он нечиняет проблем с краткими логами.

Любой каталог, содержащий файл `__init__.py`, считается пакетом Python. Каталог высшего уровня, в котором находится файл `__init__.py`, является *корневым пакетом*¹. Вложенные модули пакетов импортируются в логично 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 изпустит все его выражения верхнего уровня. Затем выполнит поиск файла с именем `pack/modu.py` и изпустит все его выражения верхнего уровня. После этих операций любая переменная, функция или класс, определенные в файле `modu.py`, будут доступны пространству имен `pack.modu`.

Спространные проблемы связаны в том, что файлы `__init__.py` содержат слишком много кода. Когда сложность проекта повышается, в структуре каталогов могут появляться подкаты и подподкаты. В этом случае импортирование одного элемента из подподката потребует запуска всех файлов `__init__.py`, встречающихся в дереве на пути к искомому.

Признаком хорошего тона является поддерживание файлов `__init__.py` пустым, когда модули и подкаты 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, существуетльтернативный корневой пакет — пакет пространства имен. Такие пакеты не должны содержать файл `__init__.py`, могут быть разбиты по нескольким каталогам `sys.path`. Python собирает все фрагменты воедино и предстает в них пользователю как один пакет.

в файле `x_init_.py`, помимо номеров версий. В проектах Tablib, Requests и Flask в этом файле есть строки документации верхнего уровня и выше лежания 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 файла 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 вводить в заблуждение, поэтому давайте проясним этот 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 программируемая логика 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 группировать инструкцию и разделение между библиотечными уровнями — наиболее пространственную причину использования языка программирования — без наличия классов.

Задачи функционального программирования (программ, которые в своей чистейшей форме не имеет операторов присваивания и побочных эффектов и вызывают функции одну за другую, чтобы выполнить задачу) могут утверждаться: из-за того, что функция выполняет работу в зависимости от состояния системы (например, от глобальной переменной, которая указывает, вошел ли пользователь под своей учетной записью), могут возникнуть ошибки и путаница. В Python (несмотря на то что он не является чисто функциональным языком) имеются инструменты, которые позволяют заложиться 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 разрешить внешние запросы, которые могут происходить одновременно. В этом случае сохранение состояния созданных объектов (означающее статичной информации о мире) может привести к *состоинию гонки*. Этот термин употребляется при описании ситуации, когда в какой-то момент между инициализацией состояния объекта (который в Python выполняется с помощью метода `Class.__init__()`) и использованием его состояния с помощью одного из методов состояния мир изменилось.

Например, запрос может загружать предмет в память и в это время поменять, что он добавлен в корзину пользователя. Если другой запрос в то же время «продолжает» тот же предмет другому человеку, может случиться, что продажа с момента произойдет после того, как первая сессия добавит предмет (в это время мы попытаемся загрузить предмет, который уже помечен как проданный). Подобные проблемы приводят к тому, что многие предпочитают функции, не сохраняющие состояния.

Мы будим следующую рекомендацию: при работе с кодом, используя некий устойчивый контекст или глобальное состояние (как в многие веб-приложения), используйте функции и процедуры, которые привнесут минимум ложное количество неявных контекстов и побочных эффектов. Неявный контекст функции создается из любых глобальных переменных и элементов на уровне сохраняемости, 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.

Пользоваться классами в Python необходимо применять для того, чтобы изолировать функции, имеющие контексты и побочные эффекты, от функций, которые имеют логику (называемые *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 пишет данные в стандартный поток вывода к побочным эффектом.

Рассмотрим преимущества чистых функций:

- их проще изменить или заменить, если нужно выполнить рефакторинг;
- их проще тестировать с помощью юнит-тестов, не нужно выполнять сложную истройку 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 инфр структур чистые функции выступают более эффективными строительными блоками, чем кл ссы или объекты, поскольку не имеют контекст и побочных эффектов. В качестве примера смотрим функции ввод-вывод, связанные с ждым форматом файла в библиотеке Tablib (`tablib/formats/*.py` — мы опишем Tablib в следующей главе). Они являются чистыми функциями, не扯остью к классам, поскольку лишь считывают данные из отдельного объект типа `Dataset`, в котором хранятся, либо записываются объекты типа `Dataset` в файл. Но объект типа `Session` в библиотеке Requests (ее мы также рассматриваем в следующей главе) — это класс, поскольку он должен сохранять cookies и информацию об аутентификации, который 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 обворачивают (или декорируют) другую функцию или метод. Декораторы не изменяют оригинальные. Поскольку функции являются объектами первого класса в Python, декораторы 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 макроиздание подойдет, чтобы изолировать основную логику функции или метода. Примером здесь, для которой нужно использовать декораторы, можно назвать поминание или кэширование: вы хотите сохранить результат дорогостоящей функции в ближайшее время 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 даёт возможность задавать значение 42, значение `thanks for all the fish`, потом устновить в качестве значения функцию.

Динамическая типизация, используемая в Python, зачастую считается недостатком, поскольку может привести к сложностям при появлению кода, для которого сложно выполнять отладку: если именованый объект может иметь в качестве значения множество различных вещей, разработчик поддерживаящий код, должен отслеживать это имя в коде, чтобы убедиться, что оно не получило неуместное значение. В таблице 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 путаницы одного имени для двух несвязанных объектов	<pre>a = 1 a = 'answer is {}'.format(a)</pre>	<pre>def get_answer(a): return 'answer is {}'.format(a) a = get_answer(1)</pre>

Продолжение ↗

Таблица 4.2 (продолжение)

Совет	Плохой код	Хороший код
Используйте разные имена для связанных элементов, если они имеют разные типы	<pre># Стока ... items = 'a b c d' # А теперь список items = items. split(' ') # А теперь множество items = set(items)</pre>	<pre>items_string = 'a b c d' items_list = items. split(' ') items = set(items_list)</pre>

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

В некоторых видах программирования, включая функциональное, рекомендуется пользоваться возможностью повторного присвоения значений переменным. В Java вы можете убедиться, что переменная всегда будет содержать одно и то же значение после присвоения, с помощью ключевого слова `final`. В Python такого ключевого слова нет (это шло бы вразрез с его философией). Но присвоение значений переменной всегда один раз может быть признаком дисциплинированности. Это помогает поддерживать 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 пользователями¹ 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 быть 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 предложены 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 хэширование¹ для хранения ключей). Неизменяемым эквивалентом списка является кортеж. Он создается добавлением круглых скобок, например `(1, 2)`. Кортеж нельзя изменить nowhere, поэтому его можно использовать как ключ словаря.

Приильное применение изменяемых типов для объектов, которые по здумке должны изменяться (например, `my_list = [1, 2, 3]`), и неизменяемых типов для объектов, которые по здумке 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 собирать все части в список, поскольку его можно изменять, тем самым объединить их. Кроме того, в Python предусмотрены *справочные включения*, которые предоставляют простой синтаксис для итерирования по входным данным для создания списка. В табл. 4.3 приведены способы создания строки из iterable-объекта.

Таблица 4.3. Способы конкатенации строки

Плохой	Хороший	Лучший
<pre>>>> s = "" >>> for c in (97, 98, 98): ... s += unichr(c) ... >>> print(s) abc</pre>	<pre>>>> s = [] >>> for c in (97, 98, 99): ... s.append(unichr(c)) ... >>> print"".join(s) abc</pre>	<pre>>>> r = (97, 98, 99) >>> s = [unichr(c) for ... c in r] >>> print"".join(s) abc</pre>

¹ Пример простого алгоритма хэширования — преобразование любого объекта в целое число и взятие его суммы по всему-нибудь модулю. memcached (<http://www.memcached.org/>) пределяет 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 контейнера известно, добавить строку будет проще (и очевиднее), чем создавать список элементов только для того, чтобы вызвать функцию `".join()`.

Все следующие варианты форматирований для определения переменной `cheese` делают одно и то же¹:

```
>>> 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, а не внутри пакета с именем `vendor` или `packages`. По адресу <http://bit.ly/on-vendorizing> вы можете прочесть весь полезный материал, в которой перечисляются основные причины, почему пакет 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 (запустив мегабайты дополнительного кода) в репозиторий. Виртуальные среды, использующие в сочетании с файлами `setup.py` (предпочтительно, особенно если пакет является библиотекой) или `requirements.txt` (при использовании переопределения зависимостей в файле `setup.py` в случае конфликтов), могут ограничивать зависимости на борту различных версий.

Если этих вариантов недостаточно, можно связаться с автором пакета для решения проблемы, обновив его пакет (например, в шести библиотек может зависеть от выходящего релиза его пакета или в нем нужна функциональность).

¹ Должны признать: хотя в PEP 3101 (<https://www.python.org/dev/peps/pep-3101/>) форматирования с использованием знаков `%` (`%s`, `%d`, `%f`) считается устаревшим, большинство разработчиков школы все еще используют его. В PEP 460 (<https://www.python.org/dev/peps/pep-0460/>) был предложен та же методика форматирования строк или объектов `bytearray`.

Эти изменения, скорее всего, пойдут на пользу всему сообществу. Однако здесь имеется и подводный камень: если вы отпривите зпрос на включение больших изменений в проект, возможно, придется поддерживать эти изменения по мере появления дальнейших предложений из просов (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

Тестируя 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 (выпущен в 2001 году), поддерживая разработку через тестирование (test-driven development, TDD): разработчик пишет тесты, которые определяют основную задачу и узкие места функций, с тем чтобы функцию, которая проходит эти тесты. С тех пор TDD становится использованием в бизнес-проектах и проектах с открытым исходным кодом — практиковаться в написании кода тест и параллельно с ним функций довольно полезно. Если пользоваться этим методом с умом, он поможет вам четко определить предзначение своего кода и создать правильную модульную структуру.

Советы по тестированию

Тест — это самый объемный фрагмент кода, который восторгает 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 заранее свежий набор данных, 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 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. Стартуйте писать тесты, которые работают быстро. Если для того, чтобы тест отработал, нужно несколько миллисекунд, разработк

будет з медлен или тесты будут з пуск ться не т к ч сто, к к в м бы этого хоте-
лось. В некоторых случ ях тесты не могут быть быстрыми, поскольку для их р боты
требуется сложн я структур д нных, котор я должен подгруж ться к ждй р з,
когд з пуск ется тест. Держите подобные тесты в отдельном н боре, который
з пуск ется к кой-нибудь з д чей по гр фику, ост льные тесты з пуск йте т к
ч сто, к к в м это нужно.

RTMF (Read the manual, friend! – «Читай руководство, друг!»). Изуч йте свои
инструменты, чтобы зн ть, к к з пустить отдельный тест или н бор тестов. При
р зр ботке функции внутри модуля поч ще з пуск йте тесты для нее, в иде ле
всякий р з, когд вы сохр няете код.

Тестируйте все в начале работы и затем опять тестируйте по ее завершении.
Всегд з пуск йте полный н бор тестов перед тем, к к пис ть код, и по з верше-
нии р боты. Это позволит убедиться, что вы ничего «не слом ли» в ост льной
ч сти код .

**Автоматические функции перехвата для системы управления версиями фанта-
стически хороши.** Ре лизов ть функцию перехв т , котор я з пуск ет все тесты
перед тем, к к отп р вить код в общий репозиторий, – хорош я идея. Вы можете
непосредственно доб влять функции перехв т в в шу систему контроля версий,
некоторые IDE предост вляют способы сдел ть это с помощью их собственных
сред. Д лее приведены ссылки н документ цию к популярным систем м контроля
версий, в которой содержится информ ция о том, к к это ре лизов ть:

- ❑ 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 ост новились н середине
сессии р зр ботки и в м нужно прерв ть р боту, можете н пис ть нер бот ющий
тест, который связ н с тем, что вы пл нируете ре лизов ть. По возвр щении к р бо-
те у в с будет ук з тель н то место, где вы ост новились (вы сможете приступить
быстрее).

В случае неопределенности выполните отладку для теста. Первый ш г отл дки
код – н пис ние еще одного тест , который ук зыв ет н ошибку. Несмотря н то
что это не всегда можно сдел ть, тесты, отл влив ющие ошибки, являются н и-
более ценными фр гмент ми код в шего проект .

Если тест сложно объяснить, то желаем вам удачи в поиске коллег. Если что-то
идет не т к или что-то нужно изменить и для в шего код н пис но множество
тестов, вы или другие сотрудники, р бот ющие н д проектом, будете пол г ться н
н бор тестов для решения проблемы или изменения поведения. Поэтому код
тест должен быть чит емым н том же уровне (или д же больше), чем р бочий код.
Юнит-тест, чье предн зн чение неясно, не принесет большой пользы.

Если тест просто объяснить, он почти всегда хорош. Код тест 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! Всё 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. Он 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 знакомо всем, кто пользуется 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 терминал в оболочке. Найдясь 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` (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, записывая сигналтуру, 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 одинковый результат (поскольку 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, в тем выполняя 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 функций 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 в том числе для каждого результата, 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 не будут 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 проектов для того, чтобы подчеркнуть правила хорошего тона при тестировании реального кода. Небольшие предполагают 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 текущего каталога в корень проекта, чтобы они указывали на то место, где лежат исходники в данном проекте, а также клонирование репозитория исходного кода и установка 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 постявляется самим. 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): ❶
    """Тестовый случай."""

    def setUp(self): ❷
        """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): ❸
        """Очистка."""
        pass
```

```

def test_empty_append(self): ④
    """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): ⑤
    """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)

```

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

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

3 Метод `TestCase.tearDown()` запускается всякий раз после каждого метода `TestCase`¹.

4 Все имена тестов должны начинаться со слов `test`, иначе они не запустятся.

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

Если вы хотите внести вклад в Tablib, первое, что можете сделать после клонирования репозитория, – это запустить набор тестов и убедиться, что все работает как положится. 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 помогает 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 командаeной 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) находитесь в тестовом слу́же, связанный с изменяемой частью проекта, и будете читать о нем при изменении кода;
б) пишете новый тестовый слу́ж для функциональности, которую хотите добавить, или для ошибки, которую отслеживаете, и будете читать о нем при изменении кода.
Рассмотрим в качестве примера следующий скрипт:

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

К сожалению, код не чист, поэтому, сначала, создайте весь набор тестов перед тем, как отправить его в репозиторий. Поскольку вы читаете о нем, 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 терминал линуксовую оболочку, перейдите во временный каталог лога, 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
```

```
=====
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 только для этого, но все еще достаточно популярны.

pytest

pytest (<http://pytest.org/latest/>) — это новый блондинок популярный модуль стандартной библиотеки Python. Это означает, что для него не требуется создавать временные файлы для тестовых случаев, возможно, даже не нужны методы установки новики и очистки. Для установки новики просто введите 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 также 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 в самом лишь нужно вызвать команду pytest. Сразу видите это сработой, которая потребуется для создания эквивалентной функции с помощью модуля unittest:

```
$ pytest
=====
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 в том числе обнаживать тесты, чтобы сэкономить время и избавить от необходимости создавать наборы тестов вручную. Предлагает множество инструментов 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 автоматизации управления средами тестирования и для тестирования в различных конфигурациях интерпретатора:

```
$ pip install tox
```

Tox позволяет сконфигурировать сложные макеты 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 несколько вариантов.

unittest2. Это общий 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 бизнесе или компании Fortune 500), можете установить его с помощью команды `pip`:

```
$ pip install unittest2
```

Вы можете захотеть импортировать модуль под именем `unittest`, чтобы в нем 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 в нем больше не потребуется модуль `unittest2`, вы сможете изменить выражение импорта, не меняя остальной код.

Mock. Если в [м](#) понр вилось то, что вы прочит ли в пункте «Mock (в модуле unittest)» р здел «Основы тестиров ния» выше, но вы р бот ете с Python в версии ниже 3.3, вы все еще можете использовать `unittest.mock`, импортиров в его к к отдельную библиотеку:

```
$ pip install mock
```

fixture. Предост вляет инструменты, которые позволяют проще и стр ив ть и очи-щ ть бэкенды б з д нных для тестиров ния (<http://farmdev.com/projects/fixture/>). Он мо-жет з груж ть ф льшивые и боры д нных для использозв ния в SQLAlchemy, SQLObject, Google Datastore, Django ORM и Storm. Существуют и его новые версии, но его тестиров ли только для версий Python 2.4-2.6.

Lettuce и Behave

Lettuce и Behave — это п кеты для выполнения *разработки через реализацию по-ве-дения* (behavior-driven development, BDD) в Python. BDD — процесс, который появился н основе TDD в н ч ле 2000-х годов для того, чтобы з менить слово «тест» в TDD н слово «поведение» (д бы преодолеть проблемы, возник ющие у новичков при освоении TDD). Это н зв ние появилось бл год ря Дэну Норту (Dan North) в 2003-м и было предст влено миру н ряду с инструментом JBehave для Java в ст тье 2006 год в журн ле *Better Socware* (предст влял собой отред ктиров ннюю ст тую из блог Дэн Норт *Introducing BDD* по дресу <http://dannorth.net/introducing-bdd>).

Концепция BDD н бр л популярность после того, к к в 2011 году вышл книга *Cucumber Book* (Pragmatic Bookshelf), где был з документиров н п кет Behave для Ruby. Это вдохновило Гэбриэл Ф лько (Gabriel Falco) н созд ние Lettuce (<http://lettuce.it/>), Питер П ренте (Peter Parente) — н созд ние Behave (<http://pythonhosted.org/behave/>) для н шего сообществ .

Поведения описыв ются простым текстом с помощью синт ксис под н зв нием Gherkin, который люди могут прочит ть, м шины — понять. В м могут приго-диться следующие руководств :

- 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

Чит емость — гл вн я цель р зр ботчиков Python к к в проект х, т к и в доку-мен-ти-ции. Приемы, опис нные в этом р зделе, помогут в м сэкономить нем ло времени.

Документация к проекту

Существует документация по API, предложенная пользователям проектов, также дополнительный 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 дать общую информацию о пользователям, кто и тем, кто обслуживает проект. В нем должен быть либо простой текст, либо легкий для чтения заметки вроде reStructured Text (сейчас это единственный формат, который понимает PyPI¹) или Markdown (<https://help.github.com/articles/basic-writing-and-formatting-syntax/>). Этот файл должен содержать несколько строк, описываящих 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 из стиля скопируются 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 «Выбирайте лицензию» далее в этой главе для получения более подробной информации.)

В файле `TODO` или одноименном разделе файл `README` должны быть представлены проблемы 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 в шаблоне документации может содержаться некоторые (или даже все) из этих компонентов:

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

¹ Для тех, кому интересно: развернулся дебаты о введении поддержки (<https://bitbucket.org/pypa/pypi/issues/148/support-markdown-for-readmes>) в файл README в PyPI.

- ❑ раздел API генерируется на основе кода (см. подраздел «Строки документации против блоковых комментариев» текущего раздела). 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 структура проектирования.

Sphinx

Sphinx (<http://sphinx.pocoo.org/>) — самый популярный¹ инструмент для создания документации для Python. Используйте его: он преобразует язык заметки reStructured Text в огромное множество форматов, включая HTML, LaTeX (для печатных 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 его. Вы можете сконфигурировать его с помощью функций перехватывающих коммитов для вашего репозитория исходного кода, поэтому перестроение в вашей документации будет происходить в том же времени.



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 пишется вся документация для Python. Если содержимое документа `long_description` функции `setuptools.setup()` написано в формате reStructured Text, оно будет отрисовано как HTML в PyPI — другие форматы будут предствлены 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 любому проекту и учитесь в процессе чтения.

¹ Вы можете встретить и другие инструменты вроде Russo, Ronn, Epydoc (больше не поддерживается) и MkDocs. Прямо как все используют Sphinx, мы также рекомендуем выбрать его.

Строки документации против блоковых комментариев

Строки документации и блоки комментариев не взаимозаменямы. Обе в рингут могут применяться для функции или класса. Рассмотрим пример использования обоих.

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

❶ Первый блок комментария — это заметка для программиста.

❷ Строки документации описывают, как работает функция или класс, он будет показан 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 в автоматически генерировать документацию с помощью привильного отформатированых строк. Инструкцию о том, как формировать 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 создан на ранней версии Python, начиная с версии 2.3. Он 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-лиза. Трекции 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 в методе нужно отобрать строку для приложения команной строки. Рассмотрим причины, почему журналирование лучше, чем `print`:

- в письме в журнале (<https://docs.python.org/library/logging.html#logrecord-attributes>), который 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 в том числе с помощью корневого средства ведения журнала в потоке журналирования для этого приложения, если только вы их не отфильтруете;
- процесс журналирования можно выбирочно приставить с помощью метода `logging.Logger.setLevel()` или отключить 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 (не библиотека) должен указывать, что случится, когда произойдет событие журналирования, мы должны конструировать:

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

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

Привилегии хорошего тона считается создание объектов средств ведения журнала только для использования вместе с переменной `_name_`: глобальный модуль журналирования создает иерархию средств ведения журнала с помощью точечной записи, поэтому использование конструкции `_name_` гарантирует отсутствие пересечений.

Рассмотрим пример применения этого приема в исходном коде библиотеки `Requests` (<https://github.com/kennethreitz/requests>) — заместите это в файле верхнего уровня `__init__.py` этого 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/>) (вторитетный 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 разместить поток событий журнала к потоку 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 файлов в формате INI	Вы можете обновлять конфигурацию при запуске функции <code>logging.config.listen()</code> , 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 словарей или файлов в формате JSON	В дополнение к обновлению во время работы вы также можете загружать конфигурацию из файла с помощью модуля <code>json</code> , 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 конфигурациями	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.python.org/3/library/logging.config.html>).

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 необязательнымиtribute ми библиотеки журналирования. Полный список доступных tribute и их описание смотрите в документации Python (<http://bit.ly/logrecord-attributes>). Предположим, что наш конфигурация журналирования называется `logging_config.ini`. Для того чтобы настроить средство 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 детелями 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 import 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 чего-либо исходного кода не указана лицензия, пользователи не получат ее безвозмездно, модифицировать или распространять ее. Помимо этого, они не смогут вносить свой вклад в проект, если вы не указете, по каким правилам играть. Поэтому в нем нужна лицензия.

Лицензии

Если в вашем проекте основанный 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 PSF¹ (при этом они сохраняют вторые 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 этим вопросом 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, начиная 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 категории².

- Разрешительные*. Часто называются похожими на Berkeley Software Distribution (BSD), больше концентрируются на том, чтобы дать пользователям свободу относительно того, что они хотят делать 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 торговыми марками без разрешения;

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](http://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 с он 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 различными ограничениями и требованиями. Все лицензии, перечисленные в табл. 4.5, ограничивают ответственность пользователей и требуют от него помнить об авторских правах на лицензии при любом распространении.

Таблица 4.5. Темы, рассматриваемые в популярных лицензиях

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

Лицензирование ресурсов

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

- ❑ GitHub предоставляет удобное руководство (<http://choosealicense.com/>), где собраны все лицензии в рамках нескольких предложений;
- ❑ на ресурсе TLDRLegal¹ (<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 и соответствие Open Source Definition (что позволит свободно использовать, модифицировать и распространять ПО).

¹ Аббревиатур tl;dr означает «Too long; didn't read» («Слишком длинно, не читал») и, скорее всего, существует как сокращение для тех, кто не стал 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 чтение 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 кода¹.

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 на языке Python;
- ❑ *Diamond* (<https://github.com/python-diamond/Diamond>) — демон Python², который собирает метрики и публикует их в Graphite или других бэкендах. Может собирать метрики для процессоров, памяти, сети, вывода-ввода, грузки и дисков. Предоставляет API для реализации пользовательских сборщиков метрик из практически любого источника;
- ❑ *Tablib* (<https://github.com/kennethreitz/tablib>) — независимая от формата библиотека, позволяющая работать с таблицами различных типов;

¹ Если вам нужны книги, материалах которой основаны 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>) в авторством Серже Демейера (Serge Demeyer), Стефана Дюассе (Stéphane Ducasse) и Оскара Нирстрасза (Oscar Nierstrasz).

² Демон — это компьютерная программа, которая работает в фоновом процессе.

- ❑ *Requests* (<https://github.com/kennethreitz/requests>) — библиотека для протокола передачи гипертекст (HyperText Transfer Protocol, HTTP) для людей (90 % из них хотят иметь HTTP-клиент, который в том числе выполняет вторизацию и соответствует многим стандартам (<https://www.w3.org/Protocols/>) для выполнения таких задач, как многокомпонентная загрузка файлов с помощью единственного вызова функции);
- ❑ *Werkzeug* (<https://github.com/mitsuhiko/werkzeug>) изначально был коллекцией различных утилит для приложений стандарта Web Service Gateway Interface (WSGI), теперь является 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 несомненно, что после прочтения текущей главы вы хотите загружать и прочесть подробнее хотя бы один-два этих проекта с постоянством (за тем, возможно, даже не скажешь о них другим).

Типичные функции

Некоторые функции одинаковы у всех проектов: есть ли снепшоты 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 необходимы, 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	10163	7	12	11	13 строк
Werkzeug	BSD 3-clause	25822	25	3	13	9 строк

В книге 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 значит, 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, хотите применить в своем коде в будущем).

HowDOI

Проект HowDOI, написанный Бенджамином Гляйтцманом (Benjamin Gleitzman), не имеет отличной отальной точки на шейодиссея, несмотря на то что он состоит менее чем из 300 строк.

Читаем сценарий, состоящий из одного файла

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

Загрузите модуль HowDOI с GitHub¹:

```
$ 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 в том числе, когда вы ввели комманду `pip install`.

Читаем документацию к HowDOI

Документация к HowDOI находится в файле `README.rst`, который содержится 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 комманду `pip uninstall lxml; pip install lxml==3.5.0`. Все работает как надо.

В ком ндной строке термин льной оболочки можно ввести ком нду `howdoi --help`, чтобы узн ть, к к пользов ться 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 документ ции мы зн ем, что HowDoI получ ет ответы н вопросы, связ нные с прогр ммиров нием, из Интернет . В руководстве ук з но, что можно выбир ть ответ в определенной позиции, р скр сить выводимую информ цию, получить несколько ответов, т кже то, что модуль имеет кэш, который можно очистить.

Использование HowDoI

Мы можем подтвердить, что поним ем, к к р бот ет HowDoI. Р ссмотрим 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 уст новили HowDoI, прочли его документ цию и теперь можем его использо в ть. Перейдем к чтению код !

Читаем код HowDoI

Если вы з глянете в к т лог `howdoi/`, то увидите дв ф ёл : `__init__.py`, который состоит всего из одной строки, ук зыв ющей номер версии, и `howdoi.py`, который мы откроем и прочит ем.

Пром трив я ф ёл `howdoi.py`, мы увидим, что к ждое новое определение функции использо вно в следующей функции; это упрощ ет чтение код . К жд я функция выполняет всего одну з д чу (он вынесен в ее имя). Гл вн я функция `command_line_runner()` расположе тся в нижней ч сти ф ёл `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 командающей строки более года, потому что они имеют одну точку входа и относительно небольшое количество путей выполнения, по которым может пойти код. (Обратите внимание, что мы вручную удалили из отрисовки много изображения функций, которые отсутствуют в проекте HowDOI, чтобы вместить график на страницу, так же немногого перерформировали его.)

Код мог бы выглядеть следующим образом: большая функция, сложенная для восприятия. Вместо этого код был разбит на отдельные функции, имеющие понятные имена. Кратко рассмотрим график, изображенный на рис. 5.1: функция `command_line_runner()` вызывает входные данные и передает файлы запроса, полученные от пользователя, в функцию `howdoi()`. Далее функция `howdoi()` обрабатывает эту функцию `_get_instructions()` в блоке `try/except`, чтобы можно было отловить ошибки соединения и вывести детальное сообщение об ошибке (поскольку код приложения не должен завершаться 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, указанных в запросе пользователя в командной строке, — по умолчанию одной).

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

Функция `_get_result()` лишь обрабатывает функцию `requests.get()` блоком `try/except`, чтобы можно было отловить ошибки SSL, вывести сообщение об ошибке и повторно генерировать исключение, чтобы блок `try/except` верхнего уровня мог отловить его и выйти (это привело хороший тон для любых программ).

Упаковка HowDOI

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

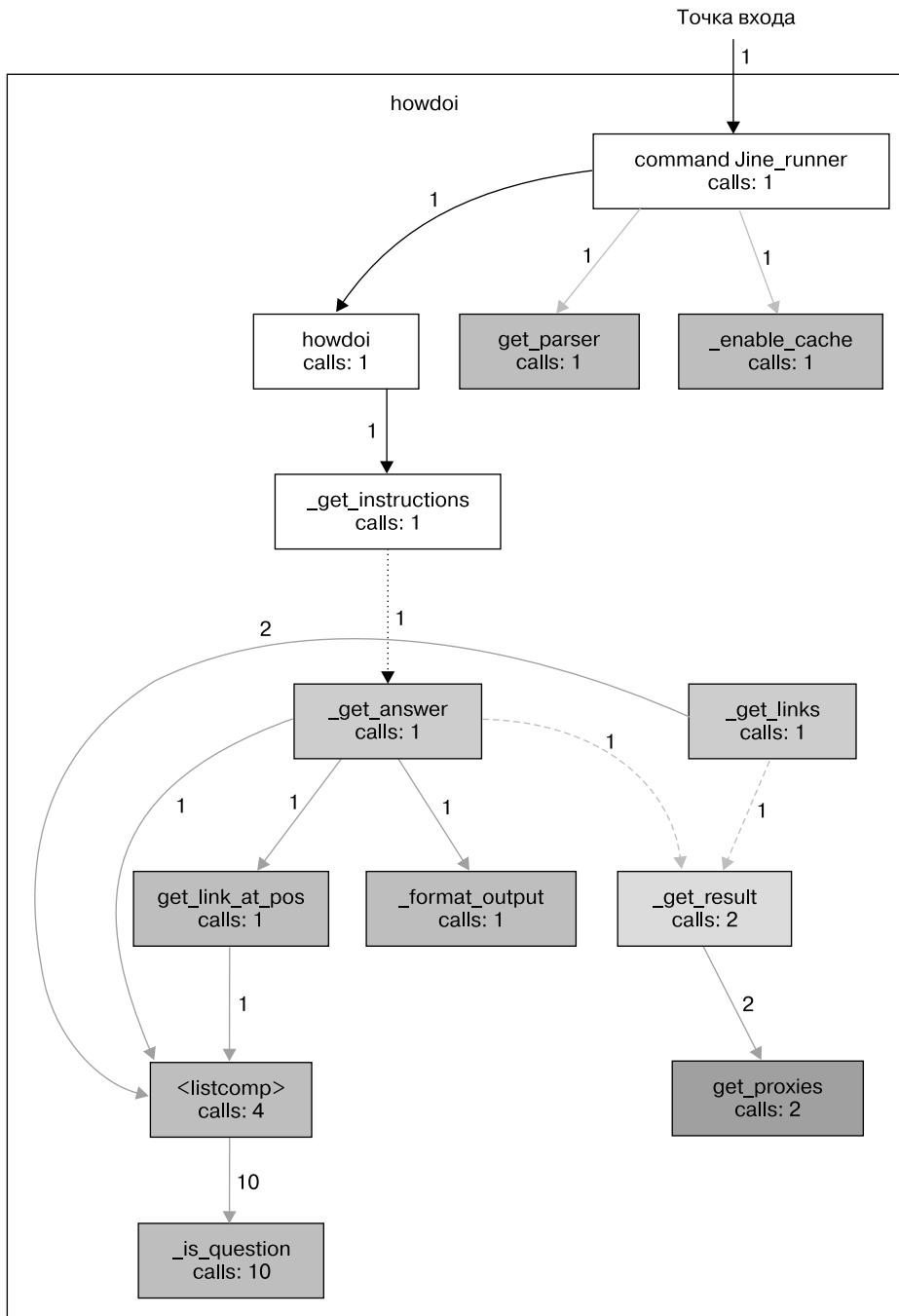


Рис. 5.1. Прозрачные пути и прозрачные имена функций в графе вызовов проекта `howdoi`

```

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, здесь лишь скажем пару слов).

Пусть каждая функция делает что-то одно

Мы не будем повторять, насколько полезно разделять внутренние функции HowDoI таким образом, чтобы они делали лишь что-то одно. Существуют функции, чье единственное предназначение — обработка других функций в операторе `try/except`. (Единственная функция, имеющая оператор `try/except`, который не следует этой практике, — `_format_output()`. Он действует операторы `try/except` не для обработки исключений, 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')
)

```

Когда мы увидим, что эти переменные существуют? Необходимость `urllib.request.getproxies()` обусловлен необязательными параметрами в `requests.get()`, поэтому часть этой информации доступна из сведений API о вызываемых в модулях. Переменные среды здесь нужны для определенной функциональности, поэтому, если библиотека пред назначена для использования с конкретной библиотекой или другим родственным приложением, в документации к этим приложениям будут перечислены соответствующие переменные среды. Для систем POSIX хорошей отправной 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 файлов, но стандартные библиотеки и внешние модули перемешаны. И несмотря на то что строковые константы в `USER_AGENTS` гораздо длиннее 80 символов, в строках нет мест, где их можно было бы разбить естественным образом, поэтому они остаются целыми.

В нескольких следующих фрагментах кода покажутся 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 ошибокного исключения (этим грешат все функции, которые вызывают функцию `_get_result()`), до тех пор пока не будет вызвана функция `howdoi()`, которая обрабатывает все возможные исключения.

Следующие внутренние функции (`_format_output()`, `_is_question()`, `_enable_cache()` и `_clear_cache()`) не предназначены для использования из пределов пакета. Тестирующий скрипт `howdoi/test_howdoi.py` вызывает только функции без префиксов, проверяя, что средство форматирования работает, и передавая аргументом одной строки для скрепления функции 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 проблема с зависимостями не возникнет, проверка версий не сорвет код в других местах. Это хорошо, поскольку `HowDOI` является конкретным инструментом одной строки и дополнительные усилия, затраченные 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 подходит 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 цитирований 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, находящиеся на 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 ③

if len(links) >= position:
    link = links[position-1] ④
else:
    link = links[-1] ⑤
return link ⑥

```

- ❶ Функция `_is_question()` определяется в отдельной строке, что указывает на четкое значение непонятному в противном случае поиску с использованием регулярных выражений.
- ❷ Списковое включение читается как предложение блоком для отдельному определению функции `_is_question()` и информативным именем переменных.
- ❸ Ранее использование оператора возврата упрощает код.
- ❹ Дополнительный шаг, который требуется на присвоение значения переменной `link`, здесь...
- ❺ ...и здесь вместо использования двух отдельных операторов возврата, не имеющих имен новых переменных, подчеркивается предположение функции `get_link_at_pos()` с помощью прозрачных имен переменных. Код с комментариями документирован.
- ❻ Единый оператор возврата, находящийся на высшем уровне отступов, явно указывает, 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 он делает. (Получив несколько ссылок и позицию, функция `get_link_at_pos()` возвращает одну ссылку, которая находится на следующей позиции.)

Diamond

Diamond — это демон (приложение, постоянно работающее в фоновый процесс), который собирает метрики системы и публикует их в формате MySQL, Graphite (<http://graphite.readthedocs.org/>) (формат с открытым исходным кодом, созданный компанией Orbitz в 2008 году, которая сохраняет, получает и по возможности строит графики на основе временных рядов) и др. У него 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 одной строки, поэтому, как и в случае с HowDoI, существуют четкие места языка и четкие пути выполнения, один из которых поддерживает код на языке в нескольких файлах.

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

Когда в случае с библиотекой HowDoI, сценарий установки Diamond добавляет исполняемые файлы `diamond` и `diamond-setup` в корневой лог `venv/bin/`. В этот раз они не генерируются в том числе, являются заранее написанными сценариями и лежат в корневом логе `Diamond/bin/`. В документации говорится, что файл `diamond` запускает сервер, `diamond-setup` – необходимый инструмент, который позволяет пользователям интерактивно изменять настройки сборщика в конфигурационном файле.

Существует множество дополнительных корневых логов, пакет `diamond` находится внутри корневого лога `Diamond/src`. Мы взглянем на файлы из корневых логов `Diamond/src` (в которых содержится основной код), `Diamond/bin` (здесь исполняемый файл `diamond`) и `Diamond/conf` (содержится пример конфигурационного файла). Остальные корневые логи 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 различных шин. Появился в 2011 году более года назад компанией BrightCove, Inc., и сегодняшний день в ее базу код внесли включено более 200 человек.

После описания истории и предыстории в документации говорится, как установить и запустить демон: в нем нужно лишь модифицировать предложенный файл конфигурации (установленный по адресу `conf/diamond.conf.example`), поместить его в соответствующее место (`/etc/diamond/diamond.conf`) или по пути, который вы указаете 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 командиной 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»), протестирував сборщики, можете изменить пользователя и группу процесса. По умолчанию он демонизирует (создает копию) процесс¹.

Используем Diamond

Для того чтобы еще лучше понять Diamond, запустим его. Для этого нужен модифицированный файл конфигурации, который мы можем поместить в созданный нами каталог `Diamond/tmp`. Начдясь 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 запустите. Недемонизированные программы вернутся в 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 работать после закрытия окна терминала. Он называется демоном в честь демона Максвелла (https://ru.wikipedia.org/wiki/Демон_Максвелла) (умного демона, не гнусного).

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

```
### Настройки для сервера
[server]
# Обработчики для опубликованных метрик ①
handlers = diamond.handler.archive.ArchiveHandler
user = ②
group =
# Каталог, откуда будут загружаться модули сборщиков ③
collectors_path = src/collectors/

### Настройки обработчиков ④
[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.

- ① Существует несколько обработчиков, каждый из которых мы можем выбрать по имени класса.
- ② Мы можем управлять пользователями и группами, под которыми запущен демон (пустое значение означает, что будут действовать текущий пользователь и группа).
- ③ Мы можем указать путь, который будет применен для поиска модулей сборщиков. Так Diamond узнает, где находятся пользовательские подключения к классам Collector (мы явно указываем это в файле конфигурации).
- ④ Конфигурацию обработчиков мы храним отдельно.

Далее пустите Diamond, указав, что журнал будет сохраняться по адресу `/dev/stdout` (будет использован стандартный поток журнала конфигурации форматированием), 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 не появится команда остановки. Журнал показвает, что делают сборщики и обработчики: сборщики собирают зарыенные метрики (вроде объем общей, свободной памяти и памяти подключки от `MemoryCollector`), которые обрабатываются форматорами и отправляются в зарыенные точки измерения, такие как Graphite, MySQL (внешнем тестовом случае — к сообщению журнала в `/dev/stdout`).

Читаем код Diamond

Для чтения более крупных проектов лучше использовать IDE — с их помощью вы можете быстро обнаружить ошибки в определении функций и классов исходного кода (при наличии определения можете найти все места в проекте, где используется функция или класс). Для использования этой функциональности укажите интерпретатор Python в вашей IDE и применять одну из виртуальных сред¹.

Вместо того чтобы разбираться в каждой функции, которую мы сделали для `HowDOI`, изучим рис. 5.2 (как это делается в импортах). Схема демонстрирует, каким образом модули `Diamond` импортируют другие модули. Подобные рисунки помогают понять более крупные проекты. Можно начать с исполняемого файла `diamond` в левом верхнем углу и следить за всеми операторами импорта в проекте `Diamond`. Помимо исполняемого файла `diamond`, в каждом каталоге также имеется файл (модуль) или пакет (пакет) в корне лога `src/diamond`.

Хорошо организованные и удобные 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 конфигурации и получения структур для сборщиков (дополнительные пути для сборщиков, определенных пользователем), `scheduler` и `signals` — для установки интервалов опроса для сборщиков, чтобы подсчитывать их метрики, а также для настройки обработчиков и указания им приступить 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 различных обработчиков, определенных в подкаталоге `handler`, и более 100 различных сборщиков, определенных в корне лога 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 путь к интерпретатору Python в текущей виртуальной среде.

`diamond.classes.load_dynamic_class()`, который я з тем вызыв ет функцию `diamond.util.load_class_from_name()` для з грузки кл ссов и основе имен, предст вленных в строк х конфигур ционного ф ёл , поэтому опер торы импорт могут не вызыв ть их явно.

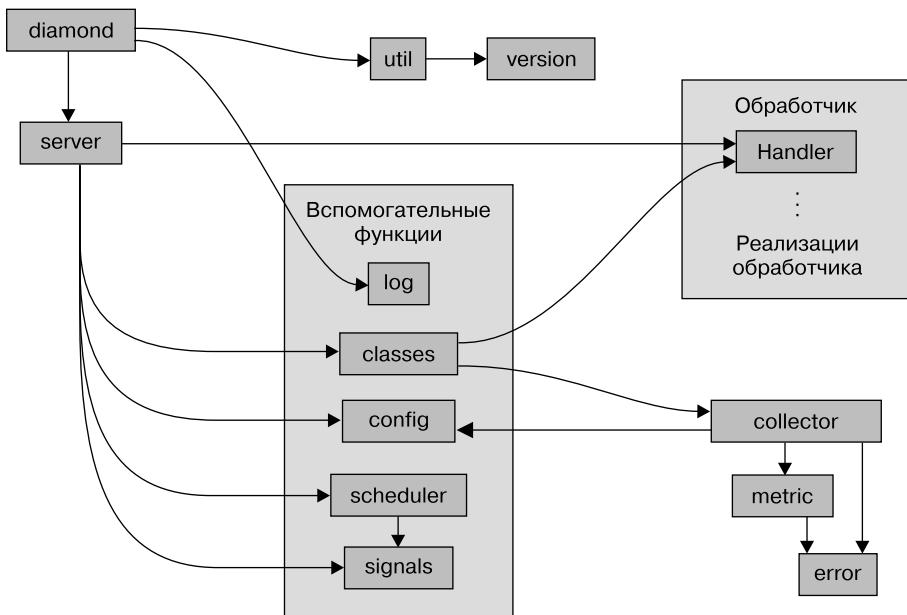


Рис. 5.2. Структура импортированных модулей в Diamond

Чтобы понять, для чего в проекте присутствуют п кет `utils` и модуль `util`, нужно открыть код: модуль `util` предоставляет функции, связанные с упаковкой Diamond (не с его р ботой): функцию для получения номер версии и основе `version.VERSION` и две функции, которые анализируют строки, позволяющие определить модули или кл ссы и импортировать их.

Журналирование в Diamond

Функция `diamond.utils.log.setup_logging()`, который я ходится в ф ёле `src/diamond/utils/log.py`, вызыв ется из функции `main()` исполняемого ф ёл `diamond` при з пуске демон :

```
# Инициализация журналирования
log = setup_logging(options.configfile, options.log_stdout)
```

Если зн чение `options.log_stdout` р вно `True`, функция `setup_logging()` и строит средство ведения журн л со ст ид ртным форм тиров нием т к, чтобы оно

отправляемые в системный 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.fileConfig()` из стандартной библиотеки Python. Перед вызовом функции — он выделен отступами, поскольку находится внутри предшествующего оператора `if/else` и блока `try/except`:

```
logging.config.fileConfig(configfile,
                           disable_existing_loggers=False)
```

Конфигурация журналов игнорирует ключевые слова в конфигурационном файле, которые не связаны с журналами. Так Diamond может использовать один и тот же конфигурационный файл как для своей конфигурации, так и для конфигурации журналов. В примере конфигурационного файла, который сполагается по адресу `Diamond/conf/diamond.conf.example`, среди прочих обработчиков определяется и обработчик журналов:

```
### Настройки обработчиков
[handlers]
# обработчик(и) для журналирования
keys = rotated_file
```

Далее в файле конфигурации под заголовком «Настройки для журналов» определяются примеры средств ведения журнала. Для получения более подробной информации смотрите документацию к конфигурационным файлам для журналов (<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 проекта, которые нам привятся, из которых 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 пока в модуль сервера, взаимодействующий с тремя другими модулями проекта: `:diamond.handler`, `:diamond.collector` и `:diamond.utils`.

В подпрограмме `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 кому надоеет ботчиков, ему воспользоваться ими. Отличный выбор!

Все реализации обработчиков содержатся в текущем логе `diamond/handler` (это логично), но структур для сборщиков отличается. Для них не предусмотрен текущий лог — только модуль `diamond/collector.py`, в котором определяются базовые классы `Collector` и `ProcessCollector`. Все реализации подклассов класса `Collector` определены в текущем логе `Diamond/src/collectors/`, в виртуальной среде они будут установлены по адресу `venv/share/diamond/collectors`, если вы установите Diamond с помощью PyPI (рекомендованый способ), или с помощью GitHub (как это сделали мы). Это помогает пользователям создавать новые реализации сборщиков: размещение всех сборщиков в одном месте упрощает их поиск для приложения, а также создание новых логичных сборщиков. В конец, каждая реализация сборщика в `Diamond/src/collectors` находится в своем текущем логе (не в отдельном файле), что позволяет разделять 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 быстрого базового класса `:diamond.collector.Collector`¹, реализовать метод `Collector.collect()` и поместить реализацию в отдельный текущий лог по адресу `venv/src/collectors/`.

Сам по себе реализация сложна, но пользователь этого не знает. Впрочем, здесь проблема тривиальна: API сборщиков, который видят пользователю, и сложный код, блокирующий которому появился подобный интерфейс.

Сложное против запутанного. Мы можем сравнить работу со сложным кодом со швейцарскими часами — они просто работают, но внутри них находится множество

¹ В Python быстрым считается класс, для которого отдельные методы не определены. Ожидается, что разработчик определит их в подклассе. В быстром базовом классе вызов этой функции генерирует исключение `NotImplementedError`. Более современный языковый стандарт — использованием модуль Python для быстрых базовых классов (`abc`) (<https://docs.python.org/3/library/abc.html>) (впервые реализован в Python 2.6), который генерирует ошибку при создании объекта незавершенного класса. Полный спецификация приведен в PEP 3119 (<https://www.python.org/dev/peps/pep-3119>).

м леньких дет лей, вз имодействующих с высокой точностью, чтобы упростить р боту с API. Использование з пут иного кода похоже на упревление с молетом — вы н верняк должны знать, что делать, чтобы не разбиться и не сгореть¹. Мы не хотим жить в мире без с молетов, но при этом желаем пользоваться ими, не вникая в тонкости их работы. Везде, где это возможно, создавайте менее сложные пользовательские интерфейсы.

Простой пользовательский интерфейс. Для того чтобы создать собственный сборщик для иных пользователей, должен создать подкласса быстрого класса `Collector`, из тем предоставивший 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 наличие `CPUCollector`, из тем, если оно не будет найдено, использует метод `diamond.collector.Collector.collect()`, что генерирует 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 — это лог `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. Задача исключением добавления спецификаций `hostname` или `hostname_method` в общие стандартные свойства (расположенные под конфигурационным файлом) или в отдельные переопределенные значения для сборщика, пока что 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 вызывает метод `utils.load_collectors()`, используя путь, указанный в `collectors_path`. Рассмотрим большую часть этой функции (мы скроем ее для удобства).

¹ Мы перевертизовали отличную статью The Difference Between Complicated and Complex Matters (<http://bit.ly/complicated-vs-complex>) Ларри Кубан (Larry Cuban), замеченного профессором Стэнфордом.

```
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): ①
        paths = paths.split(',')
        paths = map(str.strip, paths)

    load_include_path(paths) ②

    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]) ③
                for key in subcollectors:
                    ④
                    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(), ['*']) ⑤
            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) ⑥
```

```

# 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
    fqn = '.'.join([modname, attrname])
    try:
        # Загружаем класс Collector
        cls = load_dynamic_class(fqn, Collector)
        # Добавляем класс Collector
        collectors[cls.__name__] = cls
    except Exception: ⑦
        ##~~ Записываем исключение в журнал
        # и продолжаем работу...

```

Возвращаем классы Collector

return collectors

- ➊ Ребивем строку (первый вызов функции); в противном случае пути являются списком строк, содержащих пути, которые указывают места, где расположены пользовательские подклассы класса `Collector`.
- ➋ Здесь мы рекурсивно проходим по заданным путям, добавляя к текущему пути в `sys.path`, чтобы далее можно было импортировать подклассы класса `Collector`.
- ➌ Здесь выполняется рекурсия — метод `load_collectors()` вызывает сам себя¹.
- ➍ После загрузки сборщиков из подкаталогов обновите оригинальный список пользовательских сборщиков, добавив туда загруженные сборщики из этих подкаталогов.
- ➎ С момента введения Python 3.1 модуль `importlib` становится основной библиотекой Python является предпочтительным способом сделать это (с помощью модуля `importlib.import_module`; элементы `importlib.import_module` также были портированы в Python 2.7). Это позволяет, как можно прогромно импортировать модуль, используя строку с его именем.
- ➏ Теперь можно прогромно получить доступ к прибутым модуля, имея лишь строку с именем прибута.

¹ В Python предусмотрен лимит для рекурсии (максимальное количество раз, которое функция может вызвать сама себя) — он по умолчанию значительно ограничивает эти возможности (существует для того, чтобы предупредить избыточное использование рекурсии). Вы можете узнать значение этого ограничения, введя `import sys; sys.getrecursionlimit()`.

7 Метод `load_dynamic_class` здесь можно и не использовать. Он повторно импортирует модуль, проверяет, что новый класс является классом наследником, проверяет, что он является подклассом класса `Collector`, и, если это верно, — возвращает только что загруженный класс. Избыточность этого встречается в исходном коде, который пишут большие группы людей.

8 Здесь они получают имя класса для дальнейшего использования при применении этих строк из файла конфигурации (имея только строку, содержащую имя класса).

Примеры из стиля Diamond

В Diamond вы можете найти отличный пример использования языка Python, который демонстрирует все, о чем мы говорили в пункте «Замыкания с поздним связыванием» подраздела «Распространенные подводные камни» раздела «Стиль кода». Глаза 4 по поводу того, что такое поведение замыканий весьма неточно.

Пример использования замыкания (когда подводный камень все еще подводный камень). Замыкание — это функция, использующая переменные, доступные в локальной области видимости, которые в противном случае будут недоступны при вызове функции. В других языках, возможно, будет трудно разобраться и понять, но это не относится к Python, поскольку в нем функции обрабатываются также как и любые другие объекты¹. Их пример, функции могут быть переданы как аргумент, также их можно возвращать из других функций.

Рассмотрим фрагмент кода исполняемого файла `diamond`, который показывает, как работает замыкания в Python.

```
##~~ ... Пропускаем операторы импорта ... ①

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 предоставляет функции как объекты первого класса. В частности, это означает, что язык поддерживает передачу функций в качестве аргументов другим функциям, возвращающим результат других функций, присвоение им 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 ②
    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 ③
    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): ④
```

```

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.

```

- ➊ Когда мы пропускаем код, отсутствующие части описывая в комментарии, перед которым стоят две тильды (#~).
- ➋ Мы пользуемся файлом PID¹, чтобы убедиться, что демон униклен (то есть мы не разпустили его дважды случайно), также для быстрого сообщения с другими сценариями при передаче им связанных идентификаторов процессов. Этот файл также нужен для того, чтобы удостовериться, что процесс завершился нормально (поскольку в этом сценарии файл PID удаляется лишь при нормальном завершении работы).
- ➌ Весь этот код нужен для того, чтобы предоставить контекст, ведущий к ошибке. К этому моменту либо наш процесс запущен как демон (и теперь имеет другой PID), либо мы пропустим эту часть, поскольку привильный PID уже записан в файл.
- ➍ `sigint_handler()` и есть эта ошибка. Оно определяется внутри функции `main()`, не на высшем уровне, за пределами других функций, поскольку ему нужно знать, есть ли файл PID, и если да — то где.
- ➎ 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()`.
- ➏ Эта ошибка (функция `sigint_handler()`) отправляется обратно ботчику 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, сохраняя их в объекте класса `Dataset`, несколько объектов типа `Datasets` — в объекте класса `Databook`. Объекты класса `Dataset` хранятся в формате JSON, YAML, DBF и CSV (файлы в этих форматах можно импортировать), наборы данных могут быть экспортированы в формат XLSX, XLS, ODS, JSON, YAML, DBF, CSV, TSV и HTML. Библиотека Tablib выпущена Кеннетом Ритцем (Kenneth Reitz) в 2010 году, имеет интуитивный дизайн API, который для всех проектов Ритца.

Читаем небольшую библиотеку

Tablib — это библиотека, 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/>) находится с упоминаниями в документации использования, в теме которой более подробно описываются возможности библиотеки: она предоставляет объект типа `Dataset`, который имеет строки, столбцы и заголовки. Вы можете выполнять операции ввод/вывод из разных форматов для объекта типа `Dataset`. В частности, содержимое более сложные функции используются, говорится, что вы можете добавлять к строкам теги и создавать новые столбцы, которые являются функциями других столбцов.

Используем Tablib

Tablib — это библиотека, 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 способ 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/` представляются 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 документировать модули (<http://stackoverflow.com/a/2557196>). Для нас это означает, что существует еще один способ исследовать исходный код — мы можем ввести комманду `head *.py` в терминальной оболочке, находясь на верхнем уровне проекта, чтобы показать все строки документации модуля. Вот что мы увидим для библиотеки Tablib:

```
(venv)$ cd tablib
(venv)$ head *.py
==> __init__.py <== ①
"""
    Tablib.

from tablib.core import (
    Databook, Dataset, detect, import_set, import_book,
    InvalidDatasetType, InvalidDimensions, UnsupportedFormat,
    __version__
)

==> compat.py <== ②
# -*- coding: utf-8 -*-

"""
tablib.compat
~~~~~

Модуль совместимости Tablib.

"""

==> core.py <== ③
# -*- 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.

① API высшего уровня (содержимое файла `__init__.py` доступно в `tablib` после выполнения оператора `import tablib`) имеет всего девять точек входа: классы `Databook` и `Dataset` упоминаются в документации, `detect` может использоваться для определения форматов, `import_set` и `import_book` должны импортироваться динамически, последние три класса — `InvalidDatasetType`, `InvalidDimensions` и `UnsupportedFormat` — выглядят как исключения (если код следует принципам PEP 8, мы можем склоняться к тем, что эти объекты являются пользовательскими классами на основе их регистра).

② `tablib/compat.py` — это модуль совместимости. Беглый взгляд на него покажет, что он обрабатывает проблемы совместимости между Python 2 и Python 3 и логично

HowDoI, р зреш я р зные местоположения и имен к один ковым символ м, 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 леньяк я библиотек, котор я применяет многое из ширений для Sphinx.

Документация к текущей версии Sphinx находится на странице документации Tablib (<http://docs.python-tablib.org/>). Если вы хотите построить документацию самостоятельно (пользоваться Windows пон добится ком над `make` — он есть в я, но работает киндо), 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 находятся с помощью страницных блоков предствления 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 пакете 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 к каталогу `_themes/` в системный путь, установив необходимые значения свойств `html_theme_path = ['_themes']` и `html_theme = 'kr'` в `docs/conf.py`.

Для включения в шаблон кода документации API, который генерируется в том числе на основе строк документации, используйте `autoclass::`. В них нужно скопировать форматированные строки документации в Tablib, чтобы это работало:

```
.. 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. Директивы `:inherited-members:` также добавят документацию дляtributov, унаследованных от классов-предков.

Примеры из структуры Tablib

Гл ви я особенность, которую мы хотим подчеркнуть в Tablib, — отсутствие использования классов в модулях в `tablib/formats/` (это идеально иллюстрирует и не утверждение, что не нужно везде применять классы). Далее мы приведем примеры кода, демонстрирующие использование синтаксиса декоратора в Tablib, т.к. же здесь действует класс `property` (<https://docs.python.org/library/functions.html#property>) для создания унаследованныхtributov вроде ширины и высоты и т.д. Помимо этого, покажем, каким образом он регистрирует форматы файлов, чтобы избежать дубликации блочного кода для различных типов форматов (CSV, YAML и т.д.).

В последних двух подразделах мы рассмотрим, как Tablib использует зависимости, полученные от третьей стороны, с тем 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)

Код логик форматов содержит все определенные для ввод/вывод форматы файлов. Имена модулей `_csv.py`, `_tsv.py`, `_json.py`, `_yaml.py`, `_xls.py`, `_xlsx.py`, `_ods.py` и `_xlrs.py` начинаются с нижнего подчеркивания — это указывает пользователям библиотеки, что свойства не предназначены для непосредственного использования. Мы можем перейти в код логики `formats` и выполнять поиск классов и функций. Команды `grep ^class formats/*.py` показывают отсутствие определений классов, 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 ячейки для таблиц Excel;
- ❑ `export_set(dataset)` экспортитирует набор данных в заданный формат, возвращая отформатированный строку в новом формате (для Excel возвращает объект `bytes` или binaryную строку в Python 2);
- ❑ `import_set(dset, in_stream, headers=True)` заменяет содержимое набора данных содержимым входного потока;
- ❑ `export_book(databook)` экспортитирует 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 к пространствам имен (в конце концов, они же являются отличной штукой) для разделения функций вместо того, чтобы использовать тью

ненужные классы. Мы узнаем предписание к ждой функции по ее имени, например `formats._csv.import_set()`, `formats._tsv.import_set()` и `formats._json.import_set()` импортируют из борьбы других 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`, добавляя к их дескрипторами методы `__get__()`, `__set__()` или `__delete__()` (геттер, сеттер и метод удаления). Например, поискtribut `Dataset.height` приведет к ошибке функции геттера, сеттера или удаления в зависимости от контекста применения трибута. 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
```

1 Именно т к используется декор тор. В д ином случ е свойство изменяет `Dataset.height`, чтобы оно вело себя к к свойство, не к к связ ий метод. Он может р бот ть только с метод ми кл ссов.

2 Когд свойство применяется к к декор тор, трибут `height` вернет высоту `Dataset`, но вы не можете з д ть высоту множеств д иных, вызв в `Dataset.height`.

Т к выглядят трибуты `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 т к же, к к и к любому другому трибуту, но изменить его зн чение вы не можете — оно высчитыв ется н основе д иных и всегд кту льно. Т кой диз йн API весьм эргономичен: конструкцию `data.height` проще ввести н кл ви туре, чем `data.get_height()`; понятно, что озн ч ет `data.height`. Поскольку зн чение этого свойств выводится н основе д иных (зн чение свойств нельзя з д ть, для него определен только функция-геттер), можно не пережив ть, что зн чение свойств р ссинхронизируется с ре льными д иными.

Декор тор свойств можно применить только к трибут м кл ссов и только к тем кл сс м, которые н следуют от `base object object` (н пример, `class MyClass(object)`, не `class MyClass()` — в Python 3 всегд выполняется н следов ние от объект).

Один и тот же инструмент используется при созд ании API для импорт и экспорт д иных в Tablib для разных форм тов (Tablib не хр нит строку для каждого форм та). Вместо этого применяются `Dataset`- трибуты `csv`, `json` и `yaml`, они похожи н свойств `Dataset.height` и `Dataset.width`, пок з иные в предыдущем примере: вызыв ют функцию, котор я генерирует резулт т из сохр аненных д иных или преобр зует входной форм т и з тем з меняет основные д ины. Но существует только один н бор д иных.

Когд свойство `data.csv` н ходится с левой стороны зн к «р вно», вызыв ется функция-сеттер для этого свойств , котор я преобр зует `dataset` из форм т CSV. Когд свойство `data.yaml` н ходится с првой стороны зн к «р вно» или стоит отдельно, вызыв ется функция-геттер для созд ания строки в з д ином форм те н основе внутреннего н бор д иных. Р ссмотрим 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 «принадлежит» (оператор присваивания), вызывает функцию `formats.csv.import_set()`, передавая `data` в качестве первого аргумента, и строку, содержащую ингредиенты Пингловского 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 единомуtributu 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()`), `fdel` — функцию удаления данных (оставлен пустой). Далее мы увидим код, в котором программа использует эти свойства форматов.

Форматы файлов, зарегистрированные программно (не повторяйте дома)

Tablib предоставляет все подпрограммы для форматов в подкатегории `formats`. Это делается чище основной модуль `core.py` — и целый проект становится модульным;

доб влять новые форм ты ф йлов будет нетрудно. Несмотря н то что можно копиров ть фр гменты пр ктически идентичного код и импортиров ть поведение при импорте и экспорте для к ждого форм т отдельно, все форм ты *программно* з груж ются в свойств , н зв иные в честь к ждого форм т , в кл сс `Dataset`.

В следующем примере код мы выводим все содержимое ф йл `formats/__init__.py`, поскольку ф йл не т к велик и мы хотим пок з ть, к к определяется `formats.available`.

```
# -*- кодировка: utf -8 -*-
1
"""
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) 2
```

1 В этой строке интерпрет тору Python явно ук зыв ется, что ф йл имеет кодировку UTF-8¹.

2 Определение `formats.available` н ходится в ф йле `formats/__init__.py`. Его т кже можно получить с помощью функции `dir(tablib.formats)`, но приведенный выше список более прост для восприятия.

В ф йле `core.py` вместо примерно 20 (безобр зных и сложных для поддержки) повторяющихся опис ний функции для к ждого форм т код импортирует к ждый форм т прогр менно, вызыв я функцию `self._register_formats()` в конце метод `_init_()` кл сс `Dataset`. Р ссмотрим фр гмент код , в котором приводится метод `Dataset._register_formats()`.

```
class Dataset(object):
    #
    # ... опускаем документацию и некоторые определения ...
    #
```

¹ В Python 2 по умолч нию применяется ASCII, в Python 3 – UTF-8. Существует несколько разрешенных способов вз имодействия между кодировк ми, все они перечислены в PEP 263 (<https://www.python.org/dev/peps/pep-0263/>). Вы можете использовать тот, который лучше р бот ет в в шем любимом текстовом ред кторе.

```

@classmethod ①
def _register_formats(cls):
    """Добавляем свойства форматов."""
    for fmt in formats.available: ②
        try:
            try:
                setattr(cls, fmt.title,
                        property(fmt.export_set, fmt.import_set)) ③
            except AttributeError: ④
                setattr(cls, fmt.title, property(fmt.export_set)) ⑤
            except AttributeError:
                pass ⑥
        #
        # ... пропускаем другие определения ...
        #

@property ⑦
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' ⑧

Import assumes (for now) that headers exist.
"""

pass

```

- ➊ Символ `@classmethod` является декоратором (они подробно описаны в подразделе «Декораторы» подраздела «Структурируем проект» главы 4). Декоратор модифицирует метод `_register_formats()` таким образом, что он принимает перед возвращением первого аргумента класс объекта (`Dataset`), а не его экземпляра (`self`).
- ➋ Параметр `formats.available` определен в файле `formats/__init__.py` и содержит все доступные форматы.
- ➌ В этой строке `setattr` присваивает значение прибывающему 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*.
- ➍ Если свойство `fmt.import_set` не будет определено, возникнет исключение `AttributeError`.

- ➅ Если функции импорт нет, попробуйте присвоить лишь поведение экспорт .
- ➆ Если нет ни функции импорт , ни функции экспорт , не присв ив йте ничего.
- ➇ К ждый из форм тов ф ѹлов определен здесь к к свойство, имеет опис тельную строку документ ции. Строк документ ции будет сохр нен , когда функция `property()` будет вызван в точке ➃ или ➅ для присвоения дополнительного поведения.
- ➈ \t и \n — упра вляющие последовательности, которые предст вляют собой, соответственно, символ табуляции и новую строку. Все они перечислены в документ ции 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 и логичных инструментов в Java, цель которых состоит в том, чтобы управлять доступом пользователей к данным. Это идет вразрез с философией Python, которая считает, что *мы все — ответственные пользователи*. Цель применения декоратора `@property` — отделение данных от функций просмотра, связанных с данными (в этом случае с высотой, шириной и размерами форматирования). 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 — включая лог пакета `packages`, но могут в будущем быть перемещены в систему пакетов. Код лога `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 реализуется в разделе «Зависимости, полученные 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 различные 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. Ряд языковых зависимостей имеет собственную лицензию, и верхний уровень кода 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, фразмы, из которых состоит его дзен, иррациональное влияние, которое не окуз ли языки вроде ABC (<http://bit.ly/abc-to-python>), — все это заставляет стремить дружелюбие к пользователям и производительность (более подробно об оптимизации мы поговорим в разделе «Скорость» гл. 8).

Использование свойства `__slots__` в Tablib — это тот случай, когда оптимизация имеет значение. Для этого ссылка выглядит несколько странно, но доступен только для новых классов (они описаны через несколько строк ниже), 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 он сокращает отпечаток каждого объекта класса и размер одного слова (крупные объекты создают большую экономию неравенственной, для многих объектов такая экономия не стоит затраченных усилий).

Рассмотрим фрагмент 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. 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 в этом не следует 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 которой строки создаются один объект каждого 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. ...
#

```

Проблем в том, что больше не существует трибут `__dict__`, в котором хранится объекты класса `Row`, но функция `pickle.dump()` (вызывается для сериализации объектов) по умолчанию использует `__dict__` для сериализации объектов, если только не определен метод `__getstate__()`. А логично во время десериализации (процесс, который читает сериализованные байты и восстанавливает объект в памяти), если метод `__setstate__()` не определен, метод `pickle.load()` загружает динамические трибуты 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 `Tablib` — перегрузка операторов (это позволяет рассматривать модели для Python). Нестройк поведения в этих классах позволит разработчикам, использующим в 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 и более динамичных. В первом фрагменте кода показывается интерактивное применение квадратных скобок [] как для численных индексов, так и для имен столбцов, 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 ук зыв ете в кв др тных скобк х числ , этот опер тор вернет строку, котор я н ходится в з д иной позиции.
- ❷ Бл год ря этому опер тору присв ив ния с кв др тными скобк ми...
- ❸ ... вместо исходной одной оливки у в с ст новится две.
- ❹ Здесь выполняется удаление с помощью *вырезки* – 2:7 ук зыв ет н числ 2, 3, 4, 5, 6, но не 7.
- ❺ Взгляните, и сколько сокр щ ется рецепт после выполнения.
- ❻ Вы т кже можете получить доступ к столбц м с помощью имени.

В ч сти код кл сс `Dataset`, котор я определяет поведение опер тор «кв др тные скобки», пок зыв ется, к кобр б тыв ть доступ по имени столбц 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): ❶
    self._validate(value)
    self._data[key] = Row(value)

def __delitem__(self, key):
    if isinstance(key, str) or isinstance(key, unicode):
        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 в тем...

❸ ...явно вызывая исключение `KeyError`, поэтому, если вы получите доступ по имени столбца, поведение будет таким же, как и у слова `я`. Весь блок `if/else` необязателен для работы функции — если его опустить, исключение `ValueError` будет генерироваться функцией `self.headers.index(key)` в том случае, если ключ нет в `self.headers`. Единственным его предзнакомением является предложение 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 другие из них (которые хранятся в списке `x`) не будут случайно повреждены в результате присвоения.

❻ Метод `__setitem__()` может изменить одну строку, но не столбец. Это сделано наверняка, 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__()` удаляет столбец или строку, используя ту же логику, что и метод `__getitem__()`.

Для получения более подробной информации о перегрузке операторов и других особых методах смитеите 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. Он был воспринят с большим энтузиазмом годом ранее интуитивно понятному дизайну API (это значит, что API не только прост и понятен, что в нем почти не нужен документация для того, чтобы им пользоваться).

Читаем более крупную библиотеку

Библиотека Requests и много 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 в интернет-проводнике ты встречаешь ошибку 404 для того, чтобы покрыть в некую-нибудь рекламу, вы не сможете генерировать исключение `ConnectionError`.

Читаем документацию библиотеки Requests

Библиотека Requests находится в гораздо более крупном проекте, поэтому сначала просмотрите лишь заголовки разделов в документации к Requests (<http://docs.python-requests.org/>). Requests расширяет библиотеки `urllib` и `httplib`, которые вы можете найти в стандартной библиотеке Python, предоставляемая методы, выполняющие запросы HTTP. Библиотека предусматривает поддержку международных доменов и URL, атомическую декомпрессию, атомическое 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) 7230-7235¹.

¹ Если вам нужно обновить свои знания о слове `ре`, 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 в нем определение, и нажать его.

Библиотек Requests стремится покрыть все спецификации HTTP от IETF, заслуживая всего несколько функций, небор аргументов с ключевым словом и несколько классов.

Используем Requests

Когда 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  ① 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/  ② utils.py
```

1 `cacert.pem` – стандартный набор сертификатов, который используется при проверке сертификатов SSL.

2 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 программы могут получить доступ к `chardet` и `urllib3` из стандартной библиотеки.

Мы можем разобраться в происходящем более подробно выбрав имена модулей, но если нужно больше информации, просмотрите строки документации

модуля, введя `head *.py` в корне логе верхнего уровня. В следующем списке эти строки документации приводятся в сокращенном виде (не показывается `compat.py`). Исходя из его имени (он называется, как и логичный 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 перехватов Requests.
- ❑ `models.py` — содержит основные объекты, которыми пользуется Requests.
- ❑ `sessions.py` — предоставляет объект `Session` для управления истройками 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 тройственные декораторы, 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 перехватов (`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 содержится «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`;
- ❑ осталось отметить проект довольно очевиден: `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 слова), `utils.py` — вспомогательные функции.

Идея поместить модуль коммуникации в отдельный файл `adapters.py` инновационна (во всяком случае для этого разработчик). 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 них используют для определений функций.

Строки документации Requests, совместимые со Sphinx

Если вы начинете новый проект и используете Sphinx и его расширения autodoc, вам нужно отформатировать строки документации таким образом, чтобы Sphinx смог прочитать 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 привильным, неискажая 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 засомнить, он помогает пользователям писать простой и красивый код. В этом разделе сначала рассмотримся предпочтительный дизайн для более полных сообщений об ошибках и запоминающийся API, который, как мы думаем, привел к созданию модуля `requests.api`. Затем мы рассмотрим различия между объектами `requests.Request` и `urllib.request.Request` и скажем, для чего существует объект `requests.Request`.

Высокоуровневый API (очевидный способ решить задачу, желательно единственный)

Функции, определенные в файле `api.py` (за исключением `request()`), называются методами запросов HTTP¹. Все методы представляют практически одинаковые, за исключением имени метода и возможных параметров с ключевыми словами, поэтому мы будем из этого документа 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): ❶
    """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
    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 документа 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: ③
    return session.request(method=method, url=url, **kwargs)

def get(url, params=None, **kwargs): ④
    """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) ⑤
    return request('get', url, params=params, **kwargs) ⑥
```

- ❶ Функция `request()` содержит в своей сигн туре `**kwargs`. Это означает, что дополнительные аргументы с ключевым словом не генерируют исключение, т.к. это скрывает ошибки от пользователя.
- ❷ В документации, опущенной в этом фрагменте для краткости, описывается каждый аргумент с ключевым словом, с которым связано действие. Использование `**kwargs` из сигн туры вшей функции — единственный способ для пользователя скрыть, каким должно быть содержимое `**kwargs`, не заскрывая 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`, `_exit_()` — при выходе, независимо от того, закончился ли оператор или был генерировано 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 проверяет перенаправление, поэтому в этом случае устанавливается значение `True`, если пользователь не сделал этого заранее.

❶ Функция `get()` вызывает функцию `request()`, передавая в качестве первого параметра `get`. Создание функции `get` имеет два преимущества перед использованием строкового аргумента `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; он существует для того, чтобы предоставить пользователю простой API. Плюсом замещение строковых методов HTTP непосредственно в API в качестве имен функций означает, что любая опечатка в имени метода будет найдена при выполнении, 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` в качестве основного API. Зачем вообще нужен файл `models.Request`? В стандартной библиотеке уже существует `urllib.request.Request`, и в файле `cookies.py` находится объект `MockRequest`, который обворачивает `models.Request`, чтобы он работал как `urllib.request.Request` для `http.cookiejar`¹. Это означает, что любые методы, необходимые для взаимодействия объекта типа `Request` с библиотекой `cookies`, не являются исключеными 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 котором случае значение `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 списка заголовков;

¹ Модуль `http.cookiejar` ранее назывался `cookielib` в Python 2, `urllib.request.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 вызовов 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 трибуты других. Это делает все функции доступа ненужными: для получения или установки заголовков требуется лишь получить доступ к словарю `request-instance.headers`. А логично пользователь может получить или изменить строку URL: `request-instance.url`.

Объект `PreparedRequest` инициализируется пустым и заполняется при вызове метода `prepared-request-instance.prepare()`, что и полняет его реальными 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 сожалению трибуту все еще можно получить доступ непосредственно. Доступен даже `PreparedRequest._cookies`, однако нижнее подчеркивание, с которого начинается это имя, не мешает тому, что трибут не предложен для использования из пределов класса, несмотря на это (мы все — ответственные пользователи).

Этот выбор позволяет пользователям изменять объекты, но они становятся гораздо более сложными, дополнительная работа выполняется внутри `PreparedRequest`, позволяя исправить проблемы с регистром и использовать словарь вместо `CookieJar` (используйте 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 незнательными, но они позволяют создать интуитивно понятный API.

Примеры из стиля Requests

Примеры стиля из Requests демонстрируют использование множеств (по нашему мнению, о них не заслуженно забывал!), мы также взглянем на модуль `requests.status_codes module` — он здесь используется для упрощения стиля остального кода и позволяет избежать применения жестко заложенных 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 также, 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 **2**):

```
# ... 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
```

1 Спецификция **kwargs позволяет пользователю предоставить любой параметр с ключевым словом для cookie (или не предоставить их вовсе).

- ❷** Арифметик множеств! Питонск я. Прост я. Доступн я в стнд ртной библиотеке. Для слов ря функция `set()` формирует множество ключей.
- ❸** Это отличный пример того, что результирующее объединение длинной строки и две короткие более разумно. Дополнительныя переменныя error не несут никакого вреда.
- ❹** Вызов `result.update(kwargs)` обновляет словорь `result` при помощи ключ/значение, содержащимися в словоре `kwargs`, изменяя существующие пары или создавая новые, 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` предствляет собой набор из 18 позиционных аргументов и одного аргумента с ключевым словом ([rfc2109](#) по умолчанию считается равным `False`). Таким образом, 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, отдавая целый словорь.

Коды состояний (читаемость имеет значение)

Файл `status_codes.py` нужен только для того, чтобы создать объект, который может искать коды состояний поtribute. Мы сначала определим словарь в файле `status_codes.py`, затем — фрагмент кода файла `sessions.py`, в котором словорь 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 ②

#
# ... пропускаем остальные ...
#
}

codes = LookupDict(name='status_codes') ③

for code, titles in _codes.items():
    for title in titles:
        setattr(codes, title, code) ④
        if not title.startswith('\\'):
            setattr(codes, title.upper(), code) ⑤

```

1 Все эти варианты состояния OK стоят в начале ключей словаря. Задача исключением счастьливого человека (\o/) и флага (\v).

2 Установленные значения расположены на отдельной строке, поэтому, когда их в будущем будут использовать в системе контроля версий это будет четко определено.

3 `LookupDict` позволяет получать доступ к своим элементам через точку, как это показано в следующей строке.

4 `codes.ok == 200` и `codes.okay == 200`.

5 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 ошибками) 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 ошибок при этом значительно снижается.

Преобразование ключей в трибуты вместо их использования в качестве строк в словаре также снижает риск ошибок. 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 ①

class SessionRedirectMixin(object): ②
    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: ③
            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 ④
                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. ...
#

```

❶ Здесь импортируются коды состояний.

❷ Кл ссы-примеси мы опишем в пункте «Примеси (еще один отличн я штук)» подр здел «Примеры структуры из Werkzeug» следующего р здел . Примесь предост вляет методы перен пр вления для основного кл сс `Session`, который определен в том же ф йле, но не пок з н в приведенном фр гменте.

❸ Мы входим в цикл, который следует з перен пр влениями и позволяет н м получить жел емое содержимое. Вся логик цикл уд лен из этого фр гмент для кр ткости.

❹ Коды состояний, предст вленные в виде текст , гор здо чит бельнее, нежели числ (которые трудно з помнить): `codes.see_other` в противном случ е выглядел бы к к 303.

❺ `codes.found` выглядел бы к к 302, `codes.moved` — к к 301. Поэтому код ст но- вится с моз документиров нным; мы можем узн ть зн чение переменных из их имен. Мы избож ли з сорения код опеч тк ми, доб вив точечную нот цию вместо слов ря и строк (н пример, `codes.found` вместо `codes["found"]`).

Werkzeug

Для того чтобы прочесть код Werkzeug, н м нужно узн ть, к к веб-серверы общ - ются с приложениями. В следующих бз ц х мы попыт лись предст вить м кси- мально короткий обзор по этому вопросу.

Интерфейс Python для вз имодействия серверов с веб-приложениями определен в PEP 333, который н пис н Филиппом Джей Эби (Phillip J. Eby) в 2003 году¹.

¹ С тех пор PEP 333 был з менен специфик цией, в которую были доб влены дет ли, х - р ктерные для Python 3, PEP 3333. Для получения вводной информ ции рекомендуем прочесть руководство по WSGI (<http://pythonpaste.org/do-it-yourself-framework.html>) Иэн Бикинг (Ian Bicking).

В нем ук з но, к к веб-сервер (вроде Apache) общ ется с приложением Python или фреймворком.

1. Сервер будет вызыв ть приложение к ждый р з при получении з прос HTTP (н пример, GET или POST).
2. Это приложение вернет итер бельный объект, содержа щий объекты тип `bytestrings`, который сервер использует для ответ н з прос HTTP.
3. В специфик ции т кже говорится, что приложение примет дв п р метр , н - пример `webapp(environ, start_response)`. П р метр `environ` будет содерж ть д нные, связ нные с з просом, п р метр `start_response` будет функцией или другим вызыв емым объектом, который з действуется для отпр вки обр тно з головк (н пример, `('Content-type', 'text/plain')`) и состояния (н при мер, `200 OK`) н сервер.

В этом документе есть еще полдюжины стр ниц дополнительной информ ции. В середине PEP 333 вы можете увидеть впеч тляющее з явление о том, к к новый стндарт сдел ал возможным созд ние 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 хо рошей кратковременной целью для WSGI является возможность использовать любой фреймворк с любым сервером.

Четыре год спустя, в 2007 году, Армин Рон кер (Armin Ronacher) выпустил Werkzeug, н мерев ясь созд ть библиотеку WSGI, которую можно использов ть для созд ния приложений WSGI и компонентов промежуточного ПО.

Werkzeug — с мый крупный п кет из тех, что мы р ссмотрим, поэтому перечислим только несколько его проектных решений.

Читаем код инструментария

Инструмент рий (тулkit) — это н бор совместимых утилит. В случ е Werkzeug все они связ ны с приложениями WSGI. Хороший способ понять отдельные утилиты и их предн зн чение — взглянуть н юнит-тесты, именно т к мы и поступим.

Получим 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, загрузку файлов и другие утилиты и инструменты сообщества. Плюс полноценный отладчик.

Эти руководства информативны, но мы используем документацию к API вместо того, чтобы изучать компоненты библиотеки. В следующем разделе рассмотрим обертки для Werkzeug (<http://werkzeug.pocoo.org/docs/0.11/wrappers/>) и документацию 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, с тем чтобы 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 клиент будет иметь тип аргумента `response_wrapper`. В этом коде мы создаем клиентов и используем их для вызовов приложений WSGI, которые созданы ранее. Для начала заберем простое приложение WSGI (его ответ будет замещен 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.
```

Д лее используйте объект werkzeug.Response:

```
>>> client = werkzeug.Client(response_app, response_wrapper=werkzeug.Response)
>>> resp=client.get("?answer=42")
>>> print(resp.data.decode())
Hello world!
```

Кл сс werkzeug.Request предоставляет содержимое слова ря среди (аргумент environ, расположенный в wsgi_app()) в форме, удобной для пользователя, т кже декоратор для преобразования функции, которая принимает объект werkzeug.Request и возвращающий werkzeug.Response приложению WSGI:

```
>>> @werkzeug.Request.application
... def wsgi_app_using_request(request):
...     msg = "A WSGI app with:\n    method: {}\n    path: {}\n    query: {}"
...     return werkzeug.Response(
...         msg.format(request.method, request.path, request.query_string))
...
```

Он 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 указан маршрут. Рассмотрим фрагмент кода, где он используется; порядковые номера определяют блоки соответствующее ему значение.

```
>>> import werkzeug
>>> from werkzeug.routing import Map, Rule
>>>
>>> url_map = Map([
...     Rule('/', endpoint='index'), ❷
...     Rule('/<any("Robin","Galahad","Arthur")>:person',
...          endpoint='ask'), ❸
...     Rule('/<other>', endpoint='other') ❹
... ])
```

```

...     Rule('/<other>', endpoint='other') ④
...
>>> env = werkzeug.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

```

1 Объект `werkzeug.Routing.Map` предоставляет основные функции для маршрутизации. Привил соответствия применяются по порядку; первым идет выбор иного правила.

2 Если в строке-3 полнителе для привил 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', {})

```

3 В противном случае второй элементом является словарь, в котором соотнесены имена иные 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'})

```

4 Обратите внимание, что `Galahad` мог соответствовать маршруту `other`, но вместо этого ему соответствует `Lancelot`, поскольку выбирался первое соответствующее правило:

```

>>> env = werkzeug.create_environ(path='/Lancelot')
>>> urls = url_map.bind_to_environ(env)
>>> urls.match()
('other', {'other': 'Lancelot'})

```

5 Если в списке привил 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 просмотра с соответствующей конечной точкой. В следующем фрагменте кода продолжим разрывать эту линию 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: {}".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 узинуть, что делает библиотека, взглянув на ее юнит-тесты. Проблема в том, что в этом случае вы смотрите на «отдельные деревья», не на «лес», то есть исследуете страницы в приложении, предизнанные для того, чтобы гарантировать, что код не даст сбой (вместо того чтобы исследовать связи между модулями). Это годится для инструментов вроде Werkzeug, содержащего модульные слои связанные компоненты.

Поскольку мы уже знаем, как работают маршрутная и обертка 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 импортируемые 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 типов `string` и `uuid.UUID` (строк 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`. Предназначен для тестирования и нужен только потому, что в Python 2 существовало явное преобразование между строками в формате `Unicode` и `byte`, но на это нельзя положиться 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 контейнер в `werkzeug.routing.Map`,

действительно неизменяем, `MultiDict` — чтобы предотвратить несколько значений с ключами строителю URL и подтвердить, что был собран правильный URL.

7) Функция `create_environ()` предназначена для тестирования: создает среду WSGI без реального запроса HTTP.

Цель этого теста — исследование связей между модулями. Мы обнаружили, что `werkzeug.routing` лишь импортирует некоторые специальные структуры данных. Остальное ячесть юнит-тестов показвает 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 тестов 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('/qwerty/uuiop')
    assert kwargs['foo'] == ('qwerty', 'uuiop')
```

Алогично немногое импортируется в файле `werkzeug/test_wrappers.py`. В тесте показывается функциональность, доступная объекту `Request` — cookies, кодировки, аутентификация, безопасность, темпуры 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')
```

К сожалению, 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 запустить их на своем компьютере (тоx в командной строке), если установлены интерпретаторы Python. Интегрирован в GitHub, поэтому, если у вас есть файл `tox.ini` на высшем уровне репозитория, как в Werkzeug, он будет запущен 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_external=
    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 гл ве 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 затрачивая много времени.

Элегантный способ угадать тип (если реализацию легко объяснить — идея, возможно, хороша)

Вам, скорее всего, приходилось анализировать текстовые файлы и преобразовывать содержимое к нужным 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 хешей, 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 с 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** Этой статье необходима, поскольку код находится в файле `werkzeug/routing.py`, и логизируемая строка является частью URL. Здесь проверяется наличие кавычек, при обнаружении они удаляются.
- ❺** `text_type` преобразует строки в формат Unicode таким образом, что они остаются совместимыми и с Python 2, и с Python 3. Этот код практически идентичен 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>[^<]*)
    <
    (?:
        (?P<converter>[a-zA-Z_][a-zA-Z0-9_]*)
        # static rule data
        (?:(?P<args>.*?)\))
        # имя преобразователя
        \:
        # 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 предупреждаем, что присвоение переменной разных значений может приводить к появлению проблем (см. подраздел «Динамическая типизация» раздела «Структурируем проект» главы 4), но не упомянули преимущества этого присвоения. 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 на [re](https://docs.python.org/3/library/re.html) (<https://docs.python.org/3/library/re.html>).

следующую философию: «Если что-то выглядит, как утка¹, 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 было применять как обычные функции, вызов `Response.__call__()` позволяет объекту класса `Response` вызвать к нему функцию с любого места.

В последнем фрагменте используется реализация некоторых классов-примесей (каждый из них определяет какую-либо функциональность в объекте класса `Request`), которых я 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 типничем не называется, они могут быть любым вызываемым объектом: свойство не с момента создания является классом. (Вы можете склонить, что оно здумывалось к функции, поскольку его имя не начинается с прописной буквы, 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 — класса, для которого соответствующим образом определен метод `__init__()`, работая с вызываемой функцией. Кстати.

В следующем фрагменте кода содержится полное определение свойства `cached_property`, которое является подклассом класса `property`. Документация класса `cached_property` говорит сама себя. Когда это свойство будет использоваться для декорирования класса `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 слово, поскольку для него определены методы `__get__()` и `__set__()`. При получении доступа к `BaseRequest.form` в первый раз он считает различные формы (если он существует), затем пишет 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 проинтерпретировать или же для него определен пользовательский 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, который и Request. Мы изучим его интерфейс и не будем смотреть на его код. Взглянем

лишь и 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(): ❶
    return Response('Index page')

def application(environ, start_response): ❷
    path = environ.get('PATH_INFO') or '/'
    if path == '/':
        response = index() ❸
    else:
        response = Response('Not Found', status=404) ❹
    return response(environ, start_response) ❺

"""
# ... и т. д. ...

```

❶ В примере, пока в строках документации, функция `index()` вызывается в ответ на запрос HTTP. Ответом будет строка `Index page`.

❷ Этот сигнал нужен в приложениях WSGI, как указано в PEP 333/PEP 3333.

❸ Класс `Response` является подклассом `BaseResponse`, поэтому ответ представляет собой объект класса `BaseResponse`.

❹ Для ответа 404 требуется лишь устновить значение ключевого слова `status`.

❺ И в дальнейшем объект `response` является вызываемой функцией сам по себе, все соответствующие заголовки идет ли имеют зарезанные значения по умолчанию (либо переопределены, если путь отличается от `/`).

Как объект класса может быть вызваной функцией? Дело в том, что для него был определен метод `BaseRequest.__call__`. В следующем примере код мы покажем этот метод.

```
class BaseResponse(object):
    #
    # ... пропускаем все остальное ...
    #

    def __call__(self, environ, start_response): ❶
        """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) ❷
        return app_iter ❸
```

❶ Этот сигн тур позволяет создать объекты класса `BaseResponse` вызываями функциями.

❷ Здесь учтены требования к вызову приложений WSGI для функции `start_response`.

❸ А здесь возвращается итерируемый объект типа `bytes`.

Порадуйте 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 вызываямой 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 — на базе связанных трибутов. В Python, в отличие от Java, вы можете реализовать множественное наследование. Это означает, что при наследовании, при которой создаются подклассы для поддюжин различных классов одновременно, — это один из способов добавить функциональность в отдельные классы. Это похоже на пространства имен.

Подобное наследование может быть полезно во вспомогательной библиотеке вроде Werkzeug, поскольку он говорит пользователям, какие функции связанные друг с другом, какие — нет. Разработчик может быть уверен, что трибуты 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 классов-примесей иногда могут требовать наличия определенных атрибутов. Эти требования вносятся в строке документации класса-примеси.

... из файла werkzeug/wrappers.py

```
class UserAgentMixin(object): ❶

    """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) ❷

class Request(BaseRequest, AcceptMixin, ETagRequestMixin,
              UserAgentMixin, AuthorizationMixin, ❸
              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
    """
❹
```

❶ 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».

❷ `UserAgentMixin.user_agent` предполагает, что существует атрибут `self.environ`.

❸ При включении в список базовых классов для `Request` предствляемый им атрибут становится доступным с помощью вызова `Request(environ).user_agent`.

❹ Больше ничего нет — мы полностью рассмотрели определение класса `Request`. Вся функциональность предствляется базовым классом или классами-примесями. Модульными, подключаемыми и такими же великолепными, как Форд Префект.

Новые классы и object

Базовый класс `object` добавляет трибуты по умолчанию, которые подгружаются другие встроенные параметры. Классы, которые не наследуются 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 изменилось 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, 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 Роннером (Armin Ronacher). Созданный в 2009 году и был выпущен 1 марта 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 разочарован тем, что он приживался слишком медленно. Идея Werkzeug заключалась в том, чтобы отвязать WSGI от всего остального, чтобы 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 — предлагающая структуру для создания приложения WSGI²: пользовательские библиотеки предстают в виде компонентов, 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/>) находится срезе линии веб-приложения длиной семь строк, в тем 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 маршрутизацией URL. Имеет встроенные инструменты для разработки и тестирования.

Вместе с тем доступны полезные руководства, поэтому следующий шаг сделать несложно.

¹ Отсылка к Ruby on Rails, популяризировавшему веб-фреймворки, следующие стилю Django «все включено», а не стилю Flask «почти ничего не включено (пока вы не добавите иконструкции)». Django — отличный выбор в том случае, если вам нужно именно функциональность, которую предоставляет Django. Он был создан для поддержки онлайн-сети (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 з пустить пример `flaskr`, который з грузили с репозитория GitHub. В документ х говорится, что это небольшой с ют для ведения блог . Н ходясь в верхнем к т логе Flask, з пустите следующие ком нды:

```
(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 созд ние веб-приложения, поэтому он не особо отлич ется от приложений ком ндной строки Diamond и HowDoI. Вместо того чтобы привести еще одну схему, иллюстрирующую прохождение поток выполнения по функциям, мы пройдемся с помощью отл дчик по приложению-примеру `flaskr`. Для этого н м нужен pdb – отл дчик Python (н ходится в ст нд ртной библиотеке).

Для н ч л доб вьте точку ост нов в ф йле `Flaskr.py`, котор я будет активизир овн , когд поток выполнения ее достигнет, что з ст вит интерактивную сессию войти в отл дчик:

```
@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)
```

З кройте ф йл и введите `python` в ком ндной строке, д бы войти в интерактивную сессию. Чтобы не з пуск ть сервер, используйте внутренние вспомогательные про граммы для тестирования Flask – для симуляции з прос HTTP GET к к т логу /, куд мы только что поместили отл дчик:

```
>>> import flask
>>> 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 метод (`show_entries()`), где мы остновились. Строк (-> `db = get_db()`) пок зыв ет выр жение, которое будет выполнено следующим, если сд ем в еще один шт в отл дчике. Пригл щение (`Pdb`) пок зыв ет н то, что мы использую отл дчик pdb.

Мы можем подняться или опуститься по стеку¹, введя в ком ндной строке и или d соответственно. Просмотрите документ цию к pdb (<https://docs.python.org/library/pdb.html>) под з головком «Ком нды отл дчик », чтобы получить полный список ком нд, с которыми можно р бот ть. Мы т кже можем ввести имен переменных, чтобы увидеть их зн чения, и любую другую ком нду Python; мы д же можем ук з ть переменным другие зн чения до того, к к продолжим выполнение код .

Если мы поднимемся по стеку н один ш г, то увидим, что вызв ло функцию `show_entries()` (с помощью точки ост нов , которую мы только что уст новили): это объект `flask.app.Flask`, имеющий слов рь с именем `view_functions`, который соотносит строковые имен (вроде `'show_entries'`) с функциями. Мы т кже увидим, что функция `show_entries()` был вызв н с `**req.view_args`.

Мы можем узн ть, чем является `req.view_args`, в интерактивной ком ндной строке отл дчик , просто введя нужное имя (это пустой слов рь — {}, 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 отл дкой мы можем открыть ф йл исходного код и следов ть по нему. Если мы продолжим подним ться по стеку, то увидим, где вызв но 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, з пущенные интерпрет тором Python. Поэтому если функция f() вызовет функцию g(), то f() пойдет н стек первой, g() при вызове будет помещен выше f(). Когда функция g() возвр щ ет зн чение, он уд ляется из стек , и функция f() продолж ет р боту с того мест , где прерв л сь. Стек н зыв ется т к потому, что он р бот ет точно т к же, к к посудомойк , которой требуется вымыть стопку т релок: новые т релки помещ ются н верх, и в м всегд нужно н чин ть р боту 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 нами 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() ❶
            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) ❷
```

❶ Это строка 1973, определенная в отладчике.

❷ 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 и функции 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 затруднить чтение. 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 оказываемся у точки остановки, помеченной

*** Newest frame:

```
> / [... truncated path ...]/flask/examples/flaskr/flaskr.py(74)show_entries()
-> db = get_db()
(Pdb) d *** Newest frame
```

Д лее можем перейти через вызов функции с помощью ком нды `n` (`next` — «д лее») или же сдел ть миним льно возможный ш г с помощью ком нды `s` (`step` — «ш г»):

```
(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--
```

Тем н этом не з кничив ется, но демонстриров ть весь м тери л довольно уточнительно. Вот что мы узни ли.

❶ Существует объект `Flask.g`, который при более дет льном р ссмотрении ок - зыв ется глоб льным контекстом (при этом он является лок льным для объект `Flask`). Он предн знчен для хр нения соединений с б зой д нных и других устойчивых объектов вроде `cookie`, которые должны «жить» дольше, чем методы кл сс `Flask`. Использование слов ря подобным обр зом позволяет хр нить переменные з предел ми пространств имен приложения `Flask`, избег я «столкновений» имен.

❷ Функция `render_template()` н ходится в конце определения функции в модуле `flaskr.py`. Это озн ч ет, что мы, по сути, з кончили р боту — возвр щ емое зн чение отпр вляется вызыв ющей функции из объект `Flask`, который мы видели при подъеме по стеку. Поэтому мы пропустим ост льную ч сть.

Отл дчик полезно использо вть лок льно 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 н ходу (любой код Python р ботает в отл дчике), после чего вы можете продолжить выполнение код .

Журналирование во Flask

Diamond содержит пример журн лирования в приложении, Flask предоставляет кой пример в библиотеке. Если вы хотите лишь избежать предупреждений «обратчик не обижен», выполните поиск строки `logging` в библиотеке Requests (`requests/requests/__init__.py`). Но если в нем необходимо предоставить поддержку журн лирования в в шей библиотеке или фреймворке, следует воспользоваться примером, предоставляемым Flask.

Журн лирование во Flask реализовано в файле `Flask/Flask/logging.py`. В нем определяется формат строк журн ла для производств (уровень журн лирования `ERROR`) и отладки (уровень журн лирования `DEBUG`), также второй код следует советам из Twelve-Factor App (<http://12factor.net/>) по записи журн лов в потоки (которые напроявляются в потоки `wsgi.errors` или `sys.stderr` в зависимости от контекст).

Средство журн лирования добавляется в основное приложение Flask в `Flask/Flask/app.py` (в следующем фрагменте кода опущены вся ненужная 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 ④
with _logger_lock: ⑤
    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
```

1 Для каждой блокировки 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.

2 Каждый 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 средств журналирования.

3 Средство журналирования для приложения Flask изначально инициализируется как `None`, чтобы его можно было создать позднее (номер 5).

4 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.

5 Если средство журналирования еще не существует (оно было инициализировано значением `None`), используем блокировку из шага 1 и создаем его.

Примеры стиля из Flask

Большинство примеров стиля из главы 4 уже был просмотрен, поэтому для Flask мы продемонстрируем всего один — реальный пример использования 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 эту маршрутизацию URL для определения корректной функции, которая нужна для генерации ответа. Синтаксис декоратора позволяет хранить логику маршрутизации из предела самой целевой функции, поддерживая функцию `flat`, его использование интуитивно.

Делать это вовсе необязательно — он существует только для того, чтобы предоставить пользователям интуитивную 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` наследует файловую структуру для импорта блонов 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 предзначение `Flask.route` — предоставить удобный декоратор для пользователей библиотеки.

Примеры структуры из Flask

Основной посыл для обоих примеров структуры для Flask — модульность. Flask целиен привлекательно структурирован и 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 создания в объектов datetime и UUID) до классов, использующих для маршрутизации URL.

Стандарты, характерные для приложения (простое лучше, чем сложное)

Flask и Werkzeug имеют модуль `wrappers.py`. Он нужен, чтобы добавлять соответствующие значения по умолчанию во Flask, фреймворк для веб-приложений, которые будут дополнять общую библиотеку вспомогательных функций 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`, это делает стиль более предпочтительным и позволяет новому подклассу `Response` получить его имя.

❷ Возможность создать подкласс `flask.wrappers.Response`, т.к. же 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 Новые изменения в классе `Response` начинаяются. Класс `Request` изменился больше, но мы не будем покрывать его в этой главе, чтобы не раздувать ее размер.

В этой небольшой интерактивной сессии покажется, 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 в интернете пользователи Flask). Резумные значения по умолчанию делают вешение более понятным для среднестатистического пользователя.

Резумные 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 безопасности сессии, по умолчанию равными `None`. Если ключ имеет значение `null`, то приложение при попытке запуска безопасности сессии генерирует ошибку. Форсированное появление таких ошибок означает, что пользователи будут создавать собственные уникальные ключи — другими (плохими) в интернете будет либо молчаливое разрешение использования ключа сессии, равно `null`, также небезопасных способов подсчета посетителей, либо предотвращение 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 ①  
  
#: Класс, который используется для объектов ответа.  
#: Смотрите :class:`~flask.Response` для получения  
#: более подробной информации.  
response_class = Response ②  
  
#: Класс, который используется в среде Jinja.  
#: .. versionadded:: 0.11  
jinja_environment = Environment ③  
  
##~~ ... пропускаем другие определения ...  
  
url_rule_class = Rule  
test_client_class = None  
session_interface = SecureCookieSessionInterface()  
  
##~~ .. и т. д. ④
```

- ➊ Здесь можно менять пользовательский класс `Request`.
- ➋ А здесь вы можете определить пользовательский класс `Response`. Эти трибути относятся к классу `Flask` (не к объекту), имеют понятные имена, благодаря которым можно узнать их назначение.
- ➌ 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.
- ➍ 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 покажем их для того, чтобы вы поняли, что эти определения классов не должны быть доступны пользователям непосредственно, — этот выбор был явно сделан при выборе структуры, чтобы сделать пользователю больше возможностей контролировать поведение `Flask`. Когда пользователи говорят о модульности `Flask`, они имеют в виду не только применение любого бэкенда для баз данных. Подразумевается, что вы можете подключать и применять различные классы ссыек.

Вы увидели примеры качественно нового кода, который следует принципам Дзен. Мы рекомендуем взглянуть на полный код перечисленных здесь программ: лучший способ это хорошим программистом — читать отличный код. И помните: когда пишешь код становится трудно, используй исходники, Люк!¹

¹ Игрок слов `use force` — `use source`. — Примеч. перев.

6

Отправляем отличный код

В этой главе описываются правила хорошего тона, применяемые при отправке кода Python. Вы в основном создадите 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, связанных 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 составлены, поэтому мы кратко рассмотрим два способа размещения пакетов на GitHub, а также поговорим о том, как загружать код на Anaconda.org — коммерческий хостинг PyPI, запущенный Continuum Analytics.

Недостаток распространения кода с помощью PyPI или другого репозитория пакетов заключается в том, что пользователь должен понимать, какие инструменты требуют версию Python, и иметь возможность использовать инструменты вроде pip для установки других зависимостей для вящего кода.

Это приемлемо при распространении кода для других разработчиков, но не подходит для распространения приложения пользователям, не являющимся программистами. Для этого в нем нужно использовать один из инструментов, поклоненных в разделе «Задор живет в шаблоне».

¹ Ходят слухи (<https://www.pypa.io/en/latest/history/#before-2013>), что они предпочитают называть себя «министерством установки». Ник Коглан (Nick Coghlan), один из основных разработчиков для работы с PEP, связанными с упаковкой, написал в своем блоге содержательную статью о своей системе, ее истории и планах (<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 пачку в разделе «Упаковка для дистрибутивов, встроенных в Linux». Это похоже на залог, но в пакет не входит интерпретатор Python.

Наконец, мы дадим вам отличный совет в разделе «Исполняемые ZIP-файлы»: если в вашем коде находится в ZIP-архиве (файл 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 в рассмотрим наиболее важные термины (в гlosсарии 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 PostgreSQL, компилятор C или разделяемого объекта библиотеки C. Они могут быть не указаны явно, но при их отсутствии сборка прервется. Если вы строите библиотеки подобным образом, в нем может помочь семинар Пол Керер (Paul Kehrer), посвященный распространению скомпилированных модулей (<http://bit.ly/kehler-seminar>).

- **Дистрибутив.** Формат распространения пакета Python (и опционально других ресурсов и методов), имеющий форму, которая может быть установлена без дальнейшей компиляции (https://packaging.python.org/en/latest/wheel_egg/).
- **Egg.** Egg (это формат дистрибутива) представляет собой ZIP-файлы с особой структурой, содержащие методы для установки. Формат был введен более года назад в библиотеке Setuptools и является стандартом многими годами, 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 в ZIP-файлы с методами,

которые `r1p` будет использовать для установки и удаления пакетов. Файл имеет расширение `.whl` и следует соглашению по именованию (указывается платформа, сборка и использует новый интерпретатор).

Помимо установки новленного Python, для запуска обычных пакетов, содержащих лишь код, написанный на Python, не нужно ничего, кроме других библиотек, написанных исключительно на Python, — их можно загружать 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 от предыдущих версий Python, например в библиотеках C или системных исполняемых файлах. Инструменты вроде Buildout и Conda придут на помощь, когда дистрибутив становится новым даже для формата `Wheel`.

Упаковываем код

Упаковка кода дистрибутива означает создание необходимой структуры файлов, добавление требуемых файлов и определение подходящих переменных для `comform`, релевантных PEP, и текущих практик хорошего тона, описанных в разделе *Packaging and Distributing Projects* («Упаковка и распространение проектов») в *Python Packaging Guide*¹, или требований 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 высоконаправленными кatalogами пакетов, которые содержат целую библиотеку. На конец, 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 `r1p` и PyPI, но в этом менеджером пакетов по умолчанию будет `conda`, репозиторием пакетов — <http://anaconda.org/>. При сборке пакетов мы рекомендуем

¹ Надежный момент существуют два URL, которые указывают на одинаковое содержимое: <https://python-packaging-user-guide.readthedocs.org/> и <https://packaging.python.org/>.

следов ть руководству, р сположенному по дресу <http://bit.ly/building-conda>. Оно з - к ичив ется инструкциями, к к з груж ть п кеты н Anaconda.org.

Если вы созд ете библиотеку для н учных или ст тистических приложений — д же если с ми не используете Anaconda, — в м пон добится уст новить дистрибутив для Anaconda, чтобы получить доступ к широкой удитории н учных р ботников, предприним телей и пользов телей Windows, которые пользуются Anaconda для получения бин рных ф йлов без лишних з бот.

PyPI

Н дежн я вз имосвяз нн я систем инструментов вроде PyPI и pip позволяет р зр ботчик м м ксим льно легко з груж ть и уст н влив ть п кеты к к для экспериментов, т к и для созд ния крупных профессион льных систем.

Если вы пишете модуль Python с открытым исходным кодом, PyPI (<http://pypi.python.org/>), более известный к к The Cheeseshop, — походящее место для того, чтобы его р зместить¹. Если для уп ковки код вы выбир ли не PyPI, другим р зр ботчик м будет сложнее его н ити и использоватьсь к к ч сть их существующих процессов. Они будут относиться к т ким проект м с подозрением, поскольку посчит ют, что либо проектом плохо уп вляют и он не готов к релизу, либо проект з брошен.

Пр вильную и кту льную информ цию об уп ковке Python вы можете получить из руководств Python Packaging Guide, поддержив емого 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 проект PyPA по дресу <https://github.com/pypa/sampleproject> демонстрирует современные пр вил хорошего тона , к с ющиеся уп ковки проект Python. В ком-

¹ В д нный момент постепенно осуществляется переход от PyPI к Warehouse (<https://warehouse.python.org/>) (н ходится н ст дии оценки). Мы можем ск з ть, что изменился пользовательский интерфейс, но не API. Если в м любопытно, Николь Хэррис (Nicole Harris), один из р зр ботчиков PyPA, написал краткое введение к Warehouse (<http://whoisnicoleharris.com/warehouse/>).

мент риях к модулю `setup.py` (<https://github.com/pypa/sampleproject/blob/master/setup.py>) вы н йдете полезные советы и описание в жных способов упрвлении для РЕР. Общая структура файлов организована согласно требований, 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 годом PyPA хорошо работает для того, чтобы избавиться от знатной путаницы (<http://stackoverflow.com/questions/6344076>) и прекратить дискуссию (<http://stackoverflow.com/questions/3220404>) о том, какой именно способ пространения, упаковки и установки библиотек Python можно называть стандартным. pip был выбран в качестве установщик 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 версиями¹.

Каждый из этих инструментов может выполнять задачи, которые другим не под силу, для более старых систем все еще может понадобиться инструмент easy_install. В частности от PyPA, расположенной по адресу http://packaging.python.org/en/latest/pip_easy_install/, сравнивается pip и easy_install и описываются возможности каждого инструмента.

При запуске собственного кода для установки новой версии можно добиться использования команды `pip install --editable`, что позволит редактировать код без переустановки новой.

Личный PyPI

Если вы хотите установить пакеты не только из PyPI (например, из внутреннего рабочего сервера, на котором хранятся проприetary пакеты компании или пакеты, проверенные и блоггированные в штабном отделе юридической службы), можете сделать это, разместив простой сервер HTTP, который работает из корня лога, содержащего пакеты, предназначенные для установки.

Предположим, что вы хотите установить пакет, который называется `MyPackage.tar.gz`. Имеется следующая иерархия каталогов логов:

```
.
|--- 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 з пустить сервер HTTP из к т лог archive, введя следующую ком нду в оболочке:

```
$ cd archive  
$ python3 -m SimpleHTTPServer 9000
```

Это з пустит простой сервер HTTP н порту 9000 и перечислит все п кеты (в и -шем случ е MyPackage). Теперь вы можете уст новить MyPackage с помощью любого уст новщик п кетов для Python. Используйте pip в ком ндной 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 миним льный сервер, совместимый с PyPI. Он может использоваться при обслуживании и бор п кетов для easy_install или pip. Содержит полезную функцион льность вроде дминистр -тивной ком нды (-U), котор я обновит все п кеты до с мых последних версий, имеющихся в PyPI.

PyPI, размещенный на S3

Помимо вышеперечисленных в ри нтов, вы можете р зместить личный сервер PyPI н Simple Storage Service от Amazon, Amazon S3 (<https://aws.amazon.com/s3/>). Для этого в м пон добятся учетн я з пись для Amazon Web Service (AWS) и сегмент S3. Убедитесь, что вы следуете пр вил м именов ния сегмент (<http://bit.ly/rules-bucket-naming>) (вы, конечно, сможете созд ть сегмент, который н руш ет пр вил именов ния, но не получите к нему доступ). Для того чтобы использов ть в ш сегмент, сн ч л созд йте вирту льную среду н собственной м шине и уст новите требуемые сторонние библиотеки с помощью PyPI или другого источник . З тем уст новите pip2pi:

```
$ pip install git+https://github.com/wolever/pip2pi.git
```

Д лее следуйте ук з ниям ф ил README для pip2pi, где вы сможете прочесть о ком нд x pip2tgz и dir2pi. В м нужно з пустить либо эту ком нду:

```
$ pip2tgz packages/ YourPackage+
```

либо эти две:

```
$ pip2tgz packages/ -r requirements.txt  
$ dir2pi packages/
```

Теперь з груж ѹте свои ф Ѣлы. Используйте клиент вроде Cyberduck (<https://duck.sh/>), чтобы к т лог с п кет ми синхронизиров ѹть с в шим сегментом S3. Убедитесь, что вместе с новыми ф Ѣл ми и п кет ми вы з груж ѹте ф Ѣл packages/simple/index.html.

По умолч анию при з грузке новых ф Ѣлов в сегмент S3 им присв ив ѹется уровень доступа «только для пользователей». Если при попытке уст новить п кет вы получите код HTTP 403, удостоверьтесь, что пр вильно н строили уровень доступа: используйте веб-консоль от Amazon, чтобы уст новить уровень EVERYONE (разреш ет чтение всех ф Ѣлов). Уст новите в ш п кет с помощью следующей ком иды:

```
$ pip install \
--index-url=http://your-s3-bucket/packages/simple/ \
YourPackage+
```

Поддержка VCS для pip

Существует возможность получ ѹть код непосредственно из системы контроля версий с помощью инструмент рір (для этого следуйте инструкциям по адресу <http://bit.ly/vcs-support>). Этот способ — льтернатив в змещению личного PyPI. Рассмотрим пример ком иды, использующей рір для получения проекта с GitHub:

```
$ pip install git+git://git.myproject.org/MyProject#egg=MyProject
```

В этой ком иде egg нужно з менить — он носит имя к т лог в шего проекта, который вы хотите уст новить.

Замораживаем код

Фраза «з морозить код» означает «созд ѹть отдельный исполняемый п кет, который можно разместить конечным пользователям, на чьих компьютерах не уст новлен Python». Разместим исполняемый ф Ѣл или п кет содержащий код приложения, т к он 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 клиент BitTorrent (<http://www.bittorrent.com/>) (в основном они написаны на Python) делают это.

Преимущество размещения кода таким способом в том, что в ше приложение будет работать независимо от того, уст новлен 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 уст новлен из-за нее. Помимо этого, прогр аммное обеспечение конечного пользователя всегда должно иметь исполняемый формат. Файлы с расширением .py предписаны для программных инженеров и системных администраторов.

Недостаток уязвимости морозки один — он увеличивается размером дистрибутива на 2–12 МБайт. Кроме того, вам придется отписываться обновленные версии в шаблона приложения, когда будут выходить обновления безопасности к Python.

Проверяем лицензии при использовании библиотек, написанных на C

Вы должны проверять лицензии для каждого проекта, который используете, на всех уровнях деревьев зависимостей. Но мы особенно хотели бы обратить внимание на Windows, поскольку всем решениям для этой операционной системы требуется 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/>) (расшифровываясь 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 в них юнит-тесты текущего ожидания, после использования другого компилятора. Мы не чиним отходить от основной темы, поэтому проигнорируйте все, что здесь написано, если вы нечестно компилируете код C для Windows, но, например, все еще могут возникнуть некоторые 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 и общими инструментов.

Мы обратим внимание на популярные инструменты для морозки в табл. 6.1. Все они взаимодействуют со стандартной библиотекой Python. Они не могут выполнять кросс-платформенную морозку¹, поэтому вы должны проводить сборки на целевых платформах.

Инструменты перечислены в том порядке, в котором описаны в этом разделе. PyInstaller и cx_Freeze могут использоваться на всех платформах, py2app работает только для OS X, py2exe — только для Windows, 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 я функциональность для вшитых легисистем.

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 он работает 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	Д	Д	Д	Д	—
Лицензия	Модифицированная GPL	Модифицированная PSF	MIT	MIT	Zlib
Windows	Д	Д	—	—	Д
Linux	Д	Д	—	—	Д
OS X	Д	Д	Д	—	—
Eggs	Д	Д	Д	—	Д
Поддержка pkg_resources*	—	—	Д	—	Д
Режим одного файла **	Д	—	—	Д	—

- * pkg_resources (https://pythonhosted.org/setuptools/pkg_resources.html) — это отдельный модуль, построенный с Setuptools, который может использоваться для втомического поиска зависимостей. При заморозке код возникают трудности, поскольку сложно вручную находить зависимости из груженные из зависимостей статического кода PyInstaller, например, только говорит о том, что все будет хорошо, когда анализ проводится для egg-харив .
- ** Режим одного файла — это способ упаковки 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 проходит¹. По адресу <https://github.com/pyinstaller/pyinstaller/wiki/Supported-Packages> перечислены пакеты, поддерживаемые PyInstaller. Список поддерживаемых физических библиотек содержит Pillow, pygame, PyOpenGL, PyGTK, PyQt4, PyQt5, PySide (за исключением подстроек для Qt) и wxPython. Поддерживаются также инструменты NumPy, Matplotlib, Pandas и SciPy.

¹ К сожалению, мы увидим при взгляде на другие установщики, сложность заключается не в поиске и объединении пакетов совместимых библиотек, а вписывании их в C, необходимых для определенной версии библиотеки Python, в обнаружении периферических файлов конфигурации, спрайтов или особой графики, а также других файлов, которые не обнаруживаются инструментом для заморозки при исследовании 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 работе ботке кода библиотек. Конечно же предоставляет инструкции, как спрятать исходный код (<http://bit.ly/hiding-source-code>), для тех, кто создает коммерческие приложения или хочет, чтобы никто не изменил их код. Обязательно прочтите текст лицензии (проконсультируйтесь сдвояком том, 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 в этом, 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 в вашей системе совместимы: для Windows вам потребуется версия XP или выше, для систем Linux – несколько консольных приложений (справки документации, где вы можете их найти), для OS X – версия 10.7 (Lion) или выше.

Вы можете использовать Wine (эмulation 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 исполняемого файла из модуля `script.py` введите следующую команду:

```
$ pyinstaller script.py
```

Для создания оконного приложения для OS X или Windows задайте файл `--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 готовый исполняемый файл и некоторые зависимые библиотеки Python.

PyInstaller помещает все библиотеки Python, используемые в вашем 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 не строить сборку (<http://pythonhosted.org/PyInstaller/#spec-file-operation>). Вы можете сделать следующее:

- ❑ связать файлы и исполняемые файлы;
- ❑ включить библиотеки времени выполнения (файлы с расширением `.dll` или `.so`), которые PyInstaller не может определить втомически;
- ❑ добавить для исполняемого файла время 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 отредактируем файл `script.spec` можно запустить только 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 заживлять 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
```

С мой простой способ создания исполняемый файл – запустить `cxfreeze` из командной строки, но есть и другие варианты (вы можете использовать систему контроля версий), если вы пишете скрипты `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 предоставить несколько дополнительных команд (и модифицировать некоторые другие).

Данные параметры можно передавать через командную строку, скрипты установки или с помощью конфигурационного файла `setup.cfg` (<https://docs.python.org/3/distutils/configfile.html>).

Скрипты `cxfreeze-quickstart` создают простой файл `setup.py`, который может быть изменен и сохранен в системе контроля версий для будущих сборок. Рассмотрим пример сессии для скрипта 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 новой версии и мы можем изменить его в соответствии с нуждами нашего приложения. Для этого вы можете нажать в документации к cx_Freeze в разделе *distutils setup scripts* (<https://cx-freeze.readthedocs.org/en/latest/distutils.html>). Существуют также примеры сценариев `setup.py` для различных приложений с минимальной функциональностью, которые показвают, каким образом живут приложения, использующие PyQt4, Tkinter, wxPython, Matplotlib, Zope и другие библиотеки. Вы можете нажать на них в контекстном меню `samples/` исходного кода cx_Freeze (https://bitbucket.org/anthony_tuininga/cx_freeze/src): перейдите из контекстного меню в высшего уровня в `cx_Freeze/cx_Freeze/samples/`. Код там же постявляется с новой библиотекой. Вы можете получить путь, введя следующую команду:

```
$ python -c 'import cx_Freeze; print(cx_Freeze.__path__[0])'
```

Когда вы кончите 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 конкретной строки.
- ② Этот параметр модифицирует cx_Freeze, чтобы вы могли обрабатывать исполняемые файлы Windows и их зависимости.
- ③ Этот параметр модифицирует, 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 общий диск 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 в том же самом скрипте сгенерируйте файл 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 работы с другими приложениями 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 каталог перед выполнением повторной сборки. Каталог 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>) — здесь письменных переменных scan_code и load_module, возможно, нужно менять 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, зайдите более старую версию py2exe из sourceforge (<https://sourceforge.net/projects/py2exe/>). В противном случае (если у вас установлена Python 3.3+) используйте pip:

```
$ pip install py2exe
```

Руководство к py2exe (<http://www.py2exe.org/index.cgi/Tutorial>) составлено превосходно (возможно, потому что документация замещена вики-странице, не в системе контроля версий). С самым простым скриптом 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>), 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++ это может быть (или не быть) возможность просто добавить библиотеки среди выполнения для Microsoft Visual C++ вместе с вашим кодом. Если в этом это доступно, изучите инструкции по пространению DLL Visual C++ вместе с исполняемыми файлами; в противном случае вы можете предложить пользователям вашего приложения способ загрузить и установить Microsoft Visual C++ 2008 redistributable package (<http://www.py2exe.org/index.cgi/Tutorial#Step52>) или Visual C++ 2010 redistributable package (<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 дальнее время никто не поддерживает, и он не был портирован на Python 3, но ее все еще используют. Как и библиотеки cx_Freeze, py2app и py2exe, она расширяет модуль distutils, добавляя комманду `bbfreeze`. Фактически более старые версии bbFreeze были основаны на cx_Freeze. Этот библиотека может пригодиться тем, кто поддерживает легкие системы и хотел бы упаковать сборки в архивы egg, чтобы затем использовать их в своей инфраструктуре. Для установки используйте комманду `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 binaryные файлы, используйте комманду `bdist_bbfreeze` следующим образом:

```
$ bdist_bbfreeze hello.py
```

Он создаст каталог `dist` по адресу, где был запущен `bbfreeze`, в этом каталоге будут находиться интерпретатор Python и исполняемый файл с таким же именем, что и скрипт (в нашем случае `hello.py`).

Для того чтобы сгенерировать архив egg, используйте новую команду `distutils`:

```
$ python setup.py bdist_bffreeze
```

Существуют и другие варианты, например разметка сборок к скриншотам или ежедневные сборки. Для того чтобы получить больше информации об использовании библиотеки, воспользуйтесь файлом `--help`:

```
$ python setup.py bdist_bffreeze --help
```

Для более тонкой настройки залейте класс `bffreeze.Freezer` (этот способ использования `bffreeze` предпочтительный). Он имеет файлы, которые указывают, применять ли сжатие в созданным ZIP-файле, включать ли интерпретатор Python, а также какие сценарии выбрать.

Упаковка дистрибутивов в Linux

Создание дистрибутивов в ОС Linux — это, возможно, «привильный способ» распространения кода для Linux: `built distribution` похож на замороженный пакет, но не содержит интерпретатора Python, поэтому размер файла меньше примерно на 2 МБ по сравнению с замороженным¹. Если для дистрибутива выходит новое обновление безопасности для Python, в ше приложение в том числе и нечт использовать эту версию Python.

Команда `bdist_rpm` из модуля `distutils` позволяет 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 в вашем коде — это приложение, которое вы планируете распространять на других платформах, в том числе придется создать отдельную конфигурацию, необходимую для заморозки приложения в Windows и OS X. При создании и поддержке одного конфигурационного файла для любого из кросс-платформенных инструментов заморозки, описанных в разделе «Заморозка живого кода» ранее (это создаст отдельные исполняемые файлы для Windows, OS X и всех дистрибутивов Linux), нужно выполнить куда меньше работы.

¹ Некоторые могли слышать, что 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 новинкой дистрибутива, но по разным причинам, которые рассматриваются в статье, посвященной `wheels`, значительно лучше создавать дистрибутивы для отдельных платформ Linux (<http://bit.ly/python-on-wheels>), пока что в этом разделе.

Создание пакетов дистрибутивов также проблематично, если в шаблоне для той версии Python, которая в данный момент не поддерживается дистрибутивом. Вам придется скомпилировать пакеты некоторых версий Ubuntu, что им нужно добавить кое-какой функционал с помощью команды `sudo apt-add-repository` перед установкой новой ветви файлов с расширением `.deb` (это ухудшает мнение о программе). Помимо этого, вам придется поддерживать эквиваленты ленты этих инструкций для каждого дистрибутива и, что еще хуже, объяснять пользователям 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 упаковать 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 работает на многих, 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 отнести его к нему с неодобрением. Пользователи Debian могут попробовать Alien (<http://joeyh.name/code/alien/>) — написанную на Perl программу, которая преобразует файлы между формами Debian, RedHat, Stampede (.s1p) и Slackware (.tgz), но ее код не обновлялся с 2014 года, и те, кто его поддерживает, уже не используются 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

Многие знают, 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 упаковки в других форматах). 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 создадите 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 него ZIP-файл POSIX символы `#!` (называемые shebang — «шебанг») — формат ZIP это позволяет, и у вас получится независимое приложение (если Python доступен по пути, указанному 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 зависимости, вы можете ввести следующие команды в оболочке, чтобы включить в них зависимости:

```
$ cd archive
$ pip freeze | xargs pip install --target=packages
$ touch packages/__init__.py
```

Команды `xargs` принимают список аргументов командной строки от `pip freeze` и превращают его в список аргументов команды `pip`, флаг `--target=packages` отвечает за создание новой директории `packages`. Команда `touch` создает пустой файл, если такого не существует. В противном случае обновляет временную метку, установленную в ее значение в данный текущий момент времени. Структура лога будет выглядеть так:

```
.|--- archive/
|   |--- __main__.py
|   |--- packages/
|       |--- __init__.py
|       |--- dependency_one/
|       |--- dependency_two/
```

Убедитесь, что вы изменяете оператор импорта для использования к лога `packages`, который только что создан:

```
#импортируем dependency_one # не эту
import packages.dependency_one as dependency_one
```

А затем рекурсивно включите все логи 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 устали новили Python, выбрали редактор, знаете знание слов «питонский», прочитали несколько строк отличного кода Python и можете поделиться собственным кодом с остальным миром. Этим стать руководством 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.

- ❑ Гл. 7 «Взаимодействие с пользователем». Мы рассмотрим библиотеки для всех типов взаимодействия с пользователем — от консольных приложений до графических и веб-приложений.
- ❑ Гл. 8 «Управление кодом и его улучшение». Мы опишем инструменты для системного администрирования, взаимодействия с библиотеками, написанными на Си и C++, а также перечислим способы повысить скорость работы Python.
- ❑ Гл. 9 «Программные интерфейсы». Мы рассмотрим библиотеки, используемые для работы с сетью, включая синхронные, а также библиотеки для сериализации и шифрования.
- ❑ Гл. 10 «Манипуляции с данными». Мы рассмотрим библиотеки, которые предстают символьные и численные алгоритмы, графики, а также инструменты для обработки изображений и аудио.
- ❑ Гл. 11 «Хранение данных». На конец, изучим различия 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 универсальный 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 отображать и интерактивно выполнять код Python. Представляет собой интерфейс между пользователями.

Пользователи видят интерфейс клиент Jupyter — написанный на CSS, HTML и JavaScript — в браузере на клиентской машине. Клиент связывается с ядром, написанным на Python (или одном из множеств 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 заметки; похож на язык, использующийся в вики-страницах), простые заметки или исполняемый код. Сервер может быть локальным (запущенным на ноутбуке пользователя) или удаленным, к примеру 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

код C в в шей системе, к к мы обсужд ли в гл ве 2. Выполнив подготовку, вы можете уст новить Jupyter из ком ндной строки с помощью pip:

```
$ pip install jupyter
```

Последние исследов ния, посвященные использованию Jupyter для обучения (<http://bit.ly/jupyter-classroom>), пок з ли, что это эффективный и популярный способ созд в ть интер активные лекции для учеников, незн комых с прогр мирами.

Приложения командной строки

Приложения ком ндной строки (т кже их называют консольными) — это компьютерные прогр ммы, предн зн ченные для использозвания из текстового интерфейс вроде оболочки ([http://en.wikipedia.org/wiki/Shell_\(computing\)](http://en.wikipedia.org/wiki/Shell_(computing))). Они могут быть простыми ком ми (вроде pep8 или virtualenv) или интер активными (вроде pip install, pip uninstall или pip freeze) — все они имеют собственные прогр метры в дополнение к общим прогр метрам pip. К к пр вило, 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 вызов pip для перечисления компонентов, которые могут присутствовать в момент вызова приложения ком ндной строки:

① ② ③

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

- ① *Команда* — это имя вызываемого исполняемого файла .
- ② *Аргументы* следуют за ком мдой и не начиняются с дефиса . Их т кже называются прогр ами и субком ми.
- ③ *Параметры* начиняются либо с одного дефиса (для отдельных символов вроде -h), либо с двух (для слов вроде --help). Их т кже называются *флагами* или *переключателями*.

Библиотеки, перечисленные в т бл. 7.1, предоставляют разные способы назначения аргументов, т кже другие полезные инструменты для приложений ком ндной строки.

К к пр вило, сначала должны использовать инструменты стандартной библиотеки Python. Добавлять другие библиотеки в проект можно только в том случае, если в стандартной библиотеке нет нужной в функциональности.

В следующих разделах представлен более подробная информация о каждом инструменте, перечисленном в т бл. 7.1.

Таблица 7.1. Инструменты для командной строки.

Библиотека	Лицензия	Причины использовать
argparse	Лицензия PSF	<ul style="list-style-type: none"> Найдется в стандартной библиотеке. Предоставляет стандартный способ обработки аргументов и параметров
docopt	Лицензия MIT	<ul style="list-style-type: none"> Позволяет управлять форматом вспомогательного сообщения. Анализирует командную строку в соответствии с соглашениями, определенными стандартом POSIX (http://bit.ly/utility-conventions)
plac	BSD 3-clause	<ul style="list-style-type: none"> Автоматически генерирует вспомогательное сообщение на основе существующей сигнатуры функции. Анализирует аргументы командной строки в фоновом режиме, передавая их непосредственно в шейфуны
click	BSD 3-clause	<ul style="list-style-type: none"> Предоставляет декораторы для создания вспомогательного сообщения и сигнатуры (похожи на plac). Позволяет объединять несколько субкоманд. Взаимодействует с другими инструментами Flask (click не зависит от Flask, но он изначально был написан для того, чтобы помочь пользователям вместе создавать инструменты для командной строки с помощью различных инструментов для Flask, при этом ничего не разрушая, поэтому уже используется в экосистеме Flask)
clint	Лицензия Internet Software Consortium (ISC)	<ul style="list-style-type: none"> Имеет такие возможности форматирования, как изменение цвета, добавление отступов и колоночное отображение текста. Позволяет выполнить проверку типов для различных,веденных интерактивно (например, с помощью регулярного выражения, также для целого числа или пути). Дает прямой доступ к списку аргументов, предоставляет простые инструменты для фильтрации и группировки
cliff	Лицензия Apache 2.0	<ul style="list-style-type: none"> Предоставляет структурированный фреймворк для крупных проектов Python, имеющих много субкоманд. Создает интерактивную среду для использования субкоманд без дополнительного кодирования

argparse

Модуль argparse (зменяет собой предыдущий 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 инициализатора:

```
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 каждую строку и создаст словарь, в котором соотносятся все аргументы и их значения. Конструкция `action='store_true'` означает, что параметр является флагом. При наличии в командной строке он будет сохранен как `True` в словаре инициализатора.

docopt

Основной принцип `docopt` (<http://docopt.org/>) заключается в том, что документация должна быть ясной и понятной. Библиотека предоставляет одну основную команду `docopt.docopt()`, также несколько функций и классов для удобства продвинутых пользователей. Функция `docopt.docopt()` принимает инструкции в стиле POSIX, написанные для ботчика, использует их для интерпретации аргументов из командной строки и возвращает словарь со всеми аргументами и параметрами, полученными из командной строки. Также он корректно обрабатывает параметры `--help` и `--version`.

В следующем примере значение переменной `arguments` — это словарь, который имеет ключи `name`, `--capitalized` и `--num_repetitions`:

```
#!/usr/bin/env python3
"""Здороваются 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 команда 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 вызовов команды, находится в сигнатуре целевой функции. Библиотека легковесна (примерно 200 строк), обобщивет argparse (<http://docs.python.org/2/library/argparse.html>) из стандартной библиотеки и предоставляет одну основную команду plac.place(), которая получает и анализирует аргумент из сигнатуры функции, анализирует командную строку, затем вызывая функцию.

Библиотека должна быть вызвана на линии терминала аргументов командной строки (Command-Line Argument Parser, clap), имя окажется ложью, поэтому он называется Plac – clap, почти наоборот. Руководство по использованию не особенно информативно, но посмотрите, как это сделано в этом примере:

```
# hello.py
def hello(name, capitalize=False, repetitions=1):
    """Здороваются 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 каждого-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):
    """Здороваются с вами."""
    if capitalize:
        name = name.upper()
    for rep in range(repetitions):
        print('Hello {}'.format(name))

```

Помимо этого, декоратор `plac.Interpreter` предоставляет легковесный способ создания 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/>) (разработанные к Command Line-Interface Creation Kit — набор для создания интерфейсов командной строки) — помочь разработчикам создавать компонуемые интерфейсы командной строки, написав минимум возможное количество кода. Документация к Click подтверждает ее связь с `docopt`.

Функция Click — создает компонуемые системы, функция `docopt` — вручную создает симметричные интерфейсы командной строки. Эти две цели конфликтуют

друг с другом. Click мешает пользователям разрывать некоторые шаблоны для того, чтобы сделать интерфейсы командной строки уникальными. Например, вы практически не можете переформатировать вспомогательные строки.

Стандарты этой библиотеки способны удовлетворить почти все запросы разработчиков, но продвинутые пользователи могут изменить их. Кстати в случае с Plac, он не действует декораторы, чтобы привязать определения к листам функциям, которые будут их использовать, переместив из смешанных функций управление программами 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):
    """Здороваются, с капитализацией 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 команда и создает вспомогательное сообщение с помощью пользовательского языка Python, унаследовавшегося от предшествующего члена стандартной библиотеки optparse, который более совместим со стандартным POSIX, чем argparse¹.

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.
```

Рекомендуемая ценность Click заключается в ее модульности — вы можете добавить внешнюю функцию группировки, затем любую другую функцию, декорированную с помощью click, в ваш проект, и они будут субкомандами для этой команды верхнего уровня:

¹ docopt не использует ни optparse, ни argparse и для команд строк документации применяет регулярные выражения.

```
import click

@click.group() ①
@click.option('--verbose', is_flag=True)
@click.pass_context ②
def cli(ctx, verbose):
    ctx.obj = dict(verbose = verbose) ③
    if ctx.obj['verbose']:
        click.echo("Now I am verbose.")

# Функция 'hello' такая же, как раньше...
if __name__ == '__main__':
    cli() ④
```

- ①** Декоратор `group()` создает комманды верхнего уровня, которые запускаются первыми (перед вызовом любой субкомандой).
- ②** С помощью декоратора `pass_context` (опционально) передаются объекты из сгруппированных комманд в субкоманду, первый аргумент становится объектом `click.core.Context`.
- ③** Этот объект имеет специальный атрибут `ctx.obj`, который можно передавать в субкоманду, используя декоратор `@click.pass_context`.
- ④** Теперь вместо функции `hello()` вызывайте функцию, которая была декорирована `@click.group()`; в немешем 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 текущую функциональность, как простые скрипты CLI и добавление отступов, индикаторов выполнения, основанных на интерфейсе, допускает неявную обработку аргументов. Кроме того, это простое и мощное средство отображения структуры. В этом примере показаны инструменты для простых скриптов 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)): ①
        for i in range(repetitions):
            greeting = 'Hello {}'.format(colored.green(name)) ②
            puts(greeting) ③
```

```

if __name__ == '__main__':
    args = Args() ④
    # 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) ⑤
    capitalize = args.any_contain('-c')
    repetitions = int(args.value_after('--reps') or 1)
    hello(name, capitalize=capitalize, repetitions=repetitions)

```

- ➊ Использование `indent` интуитивно для менеджера контекста в операторе `with`. Флаг `quote` добавляет в начало каждой строки полуожиженную фиолетовую конструкцию `~*~`.
- ➋ Модуль `colored` имеет восемь функций для цветов, также параметр отключения скршивания.
- ➌ Функция `puts()` и логичная `print()`, обрабатывающая отступы и кавычки.
- ➍ `Args` предоставляет простые инструменты фильтрации для списка аргументов. Возвращаят еще один объект `Args`, что позволяет объединять фильтры в цепочки.
- ➎ Используйте 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 команда строк. 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 быстрых классах. В них нужно реализовать методы `cliff.command.Command` для каждого субкоманды, затем `cliff.commandmanager.CommandManager` делегирует правоильной команде. Рассмотрим минимальную версию программы `hello.py`:

```

import sys

from argparse import ArgumentParser ①
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      ②

class Hello(Command):
    """Здороваемся со всеми."""

    def get_parser(self, prog_name): ③
        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): ④
        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): ⑤
    def __init__(self):
        super(MyApp, self).__init__(
            description='Minimal app in Cliff',
            version=__version__,
            command_manager=CommandManager('named_in_setup_py'), ⑥
        )

    def main(argv=sys.argv[1:]):
        myapp = MyApp()
        return myapp.run(argv)

```

- ➊ cliff использует непосредственно `argparse.ArgumentParser` для интерфейсом ндной строки.
- ➋ Получаем версию из `setup.py` (при последнем запуске `pip install`).
- ➌ Для быстрого вызова класса требуется `get_parser()` — он должен возвращать `argparse.ArgumentParser`.
- ➍ Для быстрого вызова класса требуется `take_action()` — он запускается при вызове команды `Hello`.

- 5** В основном приложении создаётся подкласс `cliff.app.App`, отвечающий за инфраструктуру журналирования, потоки ввода/вывода и все остальное, что можно применить ко всем субкомандам.
- 6** `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 — на боры инструментов и фреймворки, которые предоставляют кнопки, полосы прокрутки, индикаторы выполнения и другие за них созданные компоненты. В конце раздела коснемся игровых библиотек.

Библиотеки виджетов

С точки зрения разработки приложений с графическими интерфейсами, *виджетами* являются кнопки, слайдеры, полосы прокрутки и другие пространственные элементы интерфейса. С их помощью нам не нужно заниматься низкоуровневым программированием вроде определения того, какая кнопка (если таковая вообще существует) находится под курсором мыши в момент клика, или разработкой с помощью оконными API, используемыми различными операционными системами.

Если вы никогда не занимались разработкой интерфейсов, то вам нужно что-то, что легко используется (чтобы вы поняли, как создавать интерфейсы). Мы рекомендуем Tkinter (она находится в стандартной библиотеке Python). В нем интересуют структуры и функции базовых инструментов, который лежит в основе библиотеки, поэтому мы сгруппируем библиотеки по тулкитам, начиная 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	<ul style="list-style-type: none"> 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)	<ul style="list-style-type: none"> Может использоваться для создания приложения для Android.

Библиотека, лежащая в основе (язык)	Библиотека Python	Лицензия	Причины использовать
			<ul style="list-style-type: none"> Имеет функциональность для работы с технологией мультиплитингом. Оптимизирован для Систем, где это возможно, использует GPU
Qt (C++)	PyQt	GNU General Public License (GPL) или Commercial	<ul style="list-style-type: none"> На всех платформах выглядит одинаково. Многие приложения и библиотеки уже являются на Qt (например, Eric IDE, Spyder и/или Matplotlib), поэтому библиотек может окажаться новленной здесь. Qt5 (нельзя использовать вместе с Qt4) предоставляет инструменты для создания приложения для Android
Qt (C++)	PySide	GNU Lesser General Public License (LGPL)	Полноценный язык менеджер для PyQt, имеющий более либеральную лицензию
GTK (C) (тулkit GIMP)	PyGObject (PyGi)	GNU Lesser General Public License (LGPL)	<ul style="list-style-type: none"> Предоставляет связывание с Python для GTK+3. Должен быть знаком всем, кто уже разрабатывал для GNOME
GTK (C)	PyGTK	GNU Lesser General Public License (LGPL)	Применяйте только в том случае, если в ваш проект уже используется PyGTK; вы должны портировать код PyGTK к PyGObject
wxWindows (C++)	wxPython	Лицензия wxWindows (модифицированная LGPL)	<ul style="list-style-type: none"> Предоставляет интуитивный внешний вид, предлагая различные оконные библиотеки для каждой платформы. Для разных платформ некоторые фрагменты кода будут отличаться
Objective C	PyObjC	Лицензия MIT	<ul style="list-style-type: none"> Предоставляет интерфейс для работы с Objective C. Придает шаблону проекту для OS X интуитивный вид. Не может использоваться на других платформах

В следующих разделах предстоит более подробно информация о различных 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/Tk¹). Поскольку модуль находится в стандартной библиотеке, он является наиболее удобным и совместимым GUI-тулкитом в настоящем списке. Tk и Tkinter доступны в большинстве платформ Unix, а также в Windows и OS X.

На ресурсе TkDocs имеется хорошее руководство по Tk и 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, графический интерактивный среды для программирования, написанный исключительно на Python. Он является частью стандартной библиотеки – вы можете запустить ее из командной строки, введя комманду `idle`, или просмотреть ее исходный код. Вы можете найти место, где он установлен, введя следующую команду в оболочке:

```
$ python -c"import idlelib; print(idlelib.__path__[0])"
```

В настоящем 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, предназначененная для разработки мультимедийных приложений с поддержкой технологии multitouch. 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 называвшийся Tool Command Language – коммандный язык инструментов) – это легковесный язык, созданный Джоном Оустерхартом (John Ousterhout) (<http://web.stanford.edu/~ouster/cgi-bin/tclHistory.php>) в начале 1990-х годов для проектирования интегральных схем.

к устройствам ввода пользователя¹ и аудио, 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 системах подобного рода как Qt и GTK. Если вы работаете с десктопным приложением 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 поста вляется и код с десятком примеров, которые покажут различные API.

Qt (произносится «кьют») (<http://qt-project.org/>) — это кроссплатформенный фреймворк, который широко применяется для разработки физического ПО, но его можно использовать и для разработки приложений без физики. Существует версия Qt5 для Android (<http://doc.qt.io/qt-5/android-support.html>). Если у вас уже установлена 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 всеми инструментами, какому из них и покупки в приложении. Qt предоставляет вам несозданные временные платформы для различных типов приложений. Об интерфейс Python для Qt, PyQt и PySide, не очень хорошо документированы, поэтому лучший вариант — воспользоваться документацией к C++ от Qt (<http://doc.qt.io/>). Кратко опишем каждый интерфейс.

- *PyQt* от компании Riverbank Computing более популярен, 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, PyQt5 — только с Qt5. (Мы предлагаем воспользоваться Docker — инструментом изоляции, который мы рассмотрели в разделе «Docker» здесь. «Инструменты изоляции» главы 3, если в действительности нужно работать с кодом с помощью сразу обеих версий, чтобы постоянно не изменять пути к библиотекам.)

¹ В дополнение к поддержке мыши может обрабатываться 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-тचскрины, использующие продукты HidTouch (<https://sourceforge.net/projects/hidtoucsuite/>) 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++, работоспособный для платформы; его вы можете использовать для сборки binарных файлов дистрибутива. Для получения более подробной информации обратитесь к руководству к PyQt4 (<https://pythonspot.com/en/pyqt4/>) и к примеру 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 к полноценной замене 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 [## GTK+](https://wiki.qt.io>Hello-World-in-PySide, которая поможет вам написать первое приложение с помощью PySide.</p>
</div>
<div data-bbox=)

Библиотека виджетов GTK+ (<http://www.gtk.org/>) (расшифровывается как GIMP Toolkit — toolkit для GIMP)¹ предоставляет API основной среды рабочего стола GNOME. Программисты могут выбирать GTK+ вместо Qt в том случае, если они предпочитают C и им удобнее смотреть исходный код GTK+ или если они уже разработали 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+). Он больше не поддерживается, и команды разработчиков рекомендуют использовать PyQt5 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+ был создан для рисования с GIMP, но стал достаточно популярным для того, чтобы пользователи хотели создать с его 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 комманду `brew install pygobject`.

wxWidgets

Философия, лежащая в основе проекта wxWidgets (<https://www.wxwidgets.org/>), заключается в том, что лучший способ придать приложению привлекательный внешний вид — применить стандартное API для каждого операционной системы. Qt и GTK+ теперь также используют другие оконные библиотеки помимо X11 за кулисами, но в Qt они быстрее, а GTK+ придает им такой внешний вид, будто вы программируете GNOME. Преимущество wxWidgets заключается в том, что вы непосредственно взаимодействуете с каждой платформой, лицензия позволяет в дальнейшем использовать ее с любыми другими платформами по-разному.

Модуль расширения, который обрабатывает wxWidgets для пользователей Python, называется wxPython (<http://wxpython.org/>). Когда-то он был самой популярной оконной библиотекой в Python, возможно, более года своей философии использовалась в стандартных инструментах для работы с интерфейсом, но теперь обходные решения в Qt и GTK+ становятся достаточно удобными. Чтобы установить модуль, перейдите на сайт <http://www.wxpython.org/download.php#stable> и загрузите соответствующий пакет для вашей ОС. Не забудьте ознакомиться с руководством для wxPython (<http://bit.ly/wxpython-getting-started>).

Objective-C

Objective-C — это proprietарный язык, используемый компанией Apple для операционных систем OS X и iOS, также вы можете получить доступ к фреймворку Сосоа для разработки приложений в OS X. В отличие от других языков, Objective-C не является кроссплатформенным; он предназначен для продуктов Apple.

PyObjC — это двухсторонний мост между языком Objective-C для OS X и Python, это означает, что не только Python будет доступен к фреймворку Сосоа для разработки приложений в OS X, но и у программистов Objective-C появится доступ к Python¹.

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 стандартно (за исключением вычислений, для которых придется использовать научные библиотеки вроде NumPy и Pandas)?



Фреймворк Соса доступен только для OS X, поэтому не выбирайте Objective-C (чрез PyObjC), если вы пишете кросс-платформенное приложение.

В м пон добится уст новить Xcode, к опис но в р зделе «Уст новк Python н Mac OS X» в гл ве 2, поскольку для PyObjC нужен компилятор. PyObjC р бот ет только со ст нд ртным дистрибутивом CPython (но не с дистрибутив ми вроде PyPy или Jython), и мы рекомендуем использовать исполняемый ф ил Python, предос твленный OS X, поскольку эт версия Python был модифициров н компонией Apple и сконфигуриров н специ льно для р боты под OS X.

Для того чтобы ук з ть в шей вирту льной среде использовать интерпрет тор для Python из в шей системы, применяйте полный путь при его вызове. Если не хотите р бот ть от лиц суперпользователя, уст новите его с помощью переключ теля `--user` (это действие сохр нит библиотеку в к т логе `$HOME/Library/Python/2.7/lib/python/site-packages/`):

```
$ /usr/bin/python -m pip install --upgrade --user virtualenv
```

Активизируйте среду, войдите в нее и уст новите PyObjC:

```
$ /usr/bin/python -m virtualenv venv
$ source venv/bin/activate
(venv)$ pip install pyobjc
```

Для этого потребуется кое-то время. PyObjC пост вляется вместе с py2app (рассматривается в подразделе «py2app» в разделе «З мор жив ем код» гл вы 6), который является инструментом для OS X, позволяющим создавать пространственные отдельные binaryные файлы приложений. Ниже 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 укзан к фреймворку, поскольку предполагает 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 разработчики 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>). Выпущен под лицензией BSD. Создан на основе pyglet, предполагая фреймворк для структурирования игры в виде набора сцен, связанных пользовательскими workflows, управляемых режиссером. Используйте ее, если вам нравится стиль «сцен-режиссер-workflow», который описан в документации по адресу <http://tinyurl.com/py-cocos2d-scenes>, или если хотите использовать pyglet для рисования и SDL2 для джойстика и аудио.

Вы можете устновить cocos2D с помощью pip. Что касается SDL2, сначала проверьте его наличие в вашем менеджере пакетов, затем загрузите с сайта <https://www.libsdl.org/>. Лучший способ начать — изучить пост вложенные примеры приложений для cocos2d (<https://github.com/los-cocos/cocos/tree/master/samples>).

- ❑ *pyglet* (<https://pypi.python.org/pypi/pyglet>). Выпущен под лицензией BSD. Представляет собой набор легковесных оболочек для OpenGL, также инструменты для представления и перемещения спрайтов по окну. Просто установите ее — в нем должно хватать 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*. Выпущен под лицензией Zlib, также под лицензией GNU GPLv2.1 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. Он недоступен в PyPI, поэтому сначала найдите ее в своем менеджере пакетов, затем, если ее нет, загрузите Pygame (<http://www.pygame.org/download.shtml>).
- ❑ *Pygame-SDL2* (<http://bit.ly/pygame-sdl2>) был недавно объявлен как попытка нового решения Pygame с бэкендом в виде SDL2. Выпущен 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 информациисмотрите руководство к PySDL2 (<http://pysdl2.readthedocs.io/en/latest/tutorial/index.html>).

Веб-приложения

Поскольку мощный язык сценариев был создан для быстрого прототипирования, а также 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 связанны со сборкой веб-приложений. С их помощью мы описали интерфейс Web Server Gateway Interface (WSGI), стандартный Python, определенный в PEP 3333, который указывает, как общются веб-серверы и веб-приложения, написанные на Python.

В этом разделе мы рассмотрим веб-фреймворки, написанные на Python, их систему блонов, 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, предоставляемого клиенту интерфейсом коду, запущенному на сервере). Большинство веб-фреймворков включает в себя шаблоны и вспомогательные программы, позволяющие выполнить краткий минимум следующий действия.

- ❑ **Маршрутизация URL.** Соотнесение входящего запроса HTTP с соответствующей функцией Python (или вызовом её функцией).
- ❑ **Обработка объектов Request и Response.** Инкапсуляция информации, полученной или отправленной пользователем.
- ❑ **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 мини-тестового сервера HTTP на машине разработчика, позволяющего ускорить разработку; часто код на серверной стороне в том числе передаётся при обновлении файлов.

Вам не нужно писать код вокруг фреймворка. Он изначально должен предоставить все, что вам требуется, при этом функциональность должна быть протестирована другими разработчиками, поэтому, если вы все еще не можете начать 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 таблице 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	<ul style="list-style-type: none"> • Предоставляет структуру — в основном заранее созданную сайт, где вы разрабатываете её логику. • Автоматически генерирует административный веб-интерфейс, где непрограммисты могут добавлять или удалять данные (например, новостные статьи).

Библиотека Python	Лицензия	Причины использовать
		<ul style="list-style-type: none"> Пост вляется вместе с инструментом для объектно-реляционного отображения (object-relational mapping, ORM)
Flask	Лицензия BSD	<ul style="list-style-type: none"> Позволяет полностью контролировать все, что находится в стеке. Предоставляет легкие декораторы, которые добавляют маршрутизацию URL для любой выбранной функции. Позволяет вставлять структуры, предоставленные Django или Pyramid
Tornado	Лицензия Apache 2.0	<ul style="list-style-type: none"> Предоставляет отличную функциональность по обработке событий — Tornado использует собственный сервер HTTP. Предоставляет способ обработки множества WebSockets (полнодуплексный, устойчивый к коммуникации с помощью TCP*) или других долголетящих соединений сразу после установки
Pyramid	Модифицированная лицензия BSD	<ul style="list-style-type: none"> Предоставляет более свободную структуру, которая зависит от временной платформой, но в меньшей степени, чем Django, позволяет использовать любой интерфейс без дополнительных или библиотек шаблонов. Основан на популярном фреймворке Zope и на Pylons

* Transmission Control Protocol (TCP) — стандартный протокол, который определяет способ, с помощью которого два компьютера могут установить соединение друг с другом.

В следующих разделах приведены более подробная информация о веб-фреймворках из главы 7.3.

Django

Django (<http://www.djangoproject.com/>) — это готовый к работе веб-фреймворк. Отличный выбор для тех, кто создает сайты, ориентированные на содержимое. Предоставляет 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 начинать с нуля.

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 симпатичные компоненты фреймворка для создания веб-приложений вроде маршрутизации URL, объекты запросов и ответов HTTP, а также шаблоны. Создание приложения с помощью Flask похоже на писание стандартного модуля Python, только к некоторым функциям прикреплены маршруты (с применением декораторов, как это показано в следующем фрагменте кода). Выглядит просто:

```
@app.route('/deep-thought')
def answer_the_question():
    return 'The answer is 42.'
```

Если вы используете Flask, ответственность за выбор других компонентов приложения (если такие 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 него 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 платформ Python (<http://bit.ly/application-dispatching>). Если же есть проблемы с подстройкой внутри приложения, используйте Flask's Blueprints («Эскизы Flask») (<http://flask.pocoo.org/docs/0.10/blueprints/>).

Tornado

Tornado (<http://www.tornadoweb.org/>) — это синхронный (упрощаемый событиями и неблокирующий, как Node.js (<https://nodejs.org/en/>)) веб-фреймворк для Python, который имеет собственный цикл событий¹. Это позволяет активно 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 предупреждение: 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 вторые спросят: «А в чем смысл?»¹, поскольку WSGI — это *синхронный* интерфейс, Tornado — *асинхронный* фреймворк.

Tornado сложнее, чем Django или Flask, и используется гораздо реже. Работает с ним только в том случае, если прирост производительности, связанный с применением синхронного фреймворка, будет стоить дополнительного времени, заслуженного на прогрессировании. Если решитесь, хорошей отправной точкой станет 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 паттернами.

Вы регистрируете временную паттерну, затем вызываете ее при создании нового проекта с помощью команды `pycreate` — аналогично команде Django `django-admin startproject project-name command`, но у вас имеются параметры для работы с различными 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 убедиться в шего босса пользователь Pyramid, взгляните на портал <https://trypyramid.com/>, где имеется большой объем информации о Pyramid.

Движки для веб-шаблонов

Большая часть приложений WSGI предписан для генерации и 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 производительность».

дин мическим содержимым, сгенерирующим приложением. Это помогает придерживаться концепции разделения обязанностей¹ — мы размещаем логику в коде, отрисовку доверяем шаблонам.

Файлы шаблонов иногда пишутся дизайнерами или фронтенд-разработчиками, и сложность строк может усложнить координирование. Поэтому смотрим правильного тона для приложений, передавших динамическое содержимое движку, чтобы и для них шаблонов.

- ❑ *Никогда зачастую лучше, чем прямо сейчас.* Файл шаблонов должно передаваться только то динамическое содержимое, которое нужно отрисовать. Постарайтесь не передавать дополнительное содержимое и в любой случай гораздо проще добавлять 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, также встречаясь с кодом Python, который будет оценен именно в шаблоне. Это может привести к неконтролируемому росту сложности и внести усложняет поиск ошибок. Мы не против такого подхода — практическость побеждает краткотематику, — просто держите себя в руках.
- ❑ *Отделяйте JavaScript от HTML.* Вместо этого необходимо смешивать шаблоны JavaScript с шаблонами HTML. Не теряйте голову: изолируйте технологии, где шаблон HTML передает переменные в код JavaScript.

Все движки шаблонов, перечисленные в таблице 7.4, принадлежат ко второму поколению, их скорость отрисовки высока², функциональность создана годами практики работы со стандартными библиотеками шаблонов.

Таблица 7.4. Движки шаблонов

Библиотека Python	Лицензия	Причины использовать
Jinja2	Лицензия BSD	<ul style="list-style-type: none"> • По умолчанию используется в Flask и поддерживается с Django. • Основан на языке шаблонов Django Template Language, в шаблоны можно добавлять лишь немного логики. • Jinja2 — это движок по умолчанию для Sphinx, Ansible и Salt (если вы используете эти инструменты, выбирайте 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 является узким местом в веб-приложениях (обычно это доступ к другим).

Библиотека Python	Лицензия	Причины использовать
Chameleon	Модифицированная лицензия BSD	<ul style="list-style-type: none"> Шаблоны сами по себе являются корректными XML/HTML. Похож на Template Attribute Language (TAL) и его derivatives
Mako	Лицензия MIT	<ul style="list-style-type: none"> Используется по умолчанию в Pyramid. Работает для повышения скорости выполнения программы — используйте, когда отрисовка блонов 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. Он работает с основанным на тексте языком шаблонов, поэтому подойдет для генерации любой информации, не только HTML.

Позволяет использовать фильтры, теги, тесты и глобальные переменные. Предоставляет возможность добавлять логику в шаблоны, что сокращает объемы кода.

Рассмотрим языковые 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 использования в комбинации с веб-сервером Tornado, описанном 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 с HTML. 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 код пок з н стр ниц сйт (`site.html`), котор яр спишает блон `base.html`. Блок `content` будет встроен втом тически 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 (фйлы с рширением `*.pt`) (<https://chameleon.readthedocs.org/>) — это реализация движка 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 реализует Page Templates и «компилирует» их в бйт-код Python для повышения скорости згрузки.

Доступен в Python 2.5 и выше (включ я 3.х и PyPy) и является одним из двух движков для отрисовки, используемых Pyramid по умолчанию (см. подробнее подр здел «Pyramid» р нее в этом р зделе). (Вторым является Mako, описаный в следующем подр зделе.)

Page Templates добавляет специальный элемент `attributes` и текстовую рзметку для вшего XML-документа: набор простых языковых конструкций позволяет упрвлять потоком документов, повторением элементов, заменой текста и переводом. Благодаря синтаксису, основному и трибуту `x`, неотрисованные шаблоны страниц предстают 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 получешь»). Это позволяет упростить взаимодействие с дизайнерами, ткже прототипирование с использованием статических фйлов. 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 `` для вставки текста досточко пространен. Если вам не нужна стопроцентная корректность для неотрисованных блонов, вы можете изменить его на более сжатый и читаемый синтаксис с помощью шаблона `${expression}`, к примеру:

```
<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 `Default Text` 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 взяты из других языков шаблонов вроде Django и Jinja2. Этот язык шаблонов по умолчанию включается в веб-фреймворке Pyramid (располагается 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>\n  
    % endfor  
  </tr>  
</%def>
```

Это текстовый язык разметки, как в Jinja2, поэтому его можно использовать для чего угодно (не только для документов XML/HTML). Чтобы отрисовать простой HTML блок, 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 вставлять большое количество логики Python в страницу. Приведены рекомендации, чтобы делать это с осторожностью (только при большой необходимости).

Развертывание веб-приложений

Давайте рассмотрим, которые мы рассматриваем в этом разделе, связанные либо с хостингом (то есть вы платите пост визуику вроде Heroku, Gondor или PythonAnywhere, чтобы они управляем сервером и бывают для других языков), либо с самой собственной инфраструктурой 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 новинка облачной вычислительной инфраструктуры, которая быстро генерирует инфраструктуру и управляет ею (например, настраивает базы данных и веб-сервер, поддерживает их с помощью обновлений, связанных с безопасностью), также отвечает за маршрутизацию и миграции приложений. При использовании PaaS разработчики приложений могут сфокусироваться на написании кода, не занимаясь управлением серверами и инфраструктурой.

Существует множество поставщиков PaaS, конкурирующих друг с другом, но поставщики, представленные в этом списке, являются наиболее популярными в сообществе Python. Большинство предложений имеют 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. Вместе же предоставляется набор инструментов для командной строки (<https://toolbelt.heroku.com/>), чтобы работать с учетной записью Heroku, без необходимости писать скрипты. Heroku предлагает подробные статьи (<https://devcenter.heroku.com/categories/python>), посвященные использованию Python на Heroku, а также пошаговые инструкции (<https://devcenter.heroku.com/articles/getting-started-with-python>) о том, как строить ваше первое приложение.
- ❑ *Gondor* (<https://gondor.io/>). Поддерживается небольшой компанией, концентрируется на том, чтобы помочь бизнесу добиваться успеха с помощью Python и Django. Его платформа предназначена для развертывания приложений Django и Pinax¹. Платформой Gondor являются Ubuntu 12.04 и версии Django 1.4, 1.6, и 1.7, а также подмножество различных Python 2 и 3, перечисленных по адресу <https://gondor.io/support/runtime-environment/>. Он может в том числе сконфигурировать ваш сайт, написанный с помощью Django, если вы используете local_settings.py для размещения хактерной 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 таким образом, чтобы укачивать в PEP 3333, для того чтобы получить запрос HTTP и отправить ответ HTTP.

Большинство синтетических замещаемых приложений Python сегодня находится на сервере WSGI вроде Gunicorn (сервер WSGI, который можно использовать как

¹ Pinax объединяет популярные шаблоны, приложения и инфраструктуру Django, чтобы ускорить запуск проекта Django.

отдельные сервер HTTP) или на базе легковесного веб-сервера вроде Nginx. Если в действуются обработчики WSGI, серверы WSGI взаимодействуют с приложениями Python в то время, как веб-сервер обрабатывает более подходящие задачи — выдачу статических файлов, маршрутизацию просов, защиту от пределенных атак и т. п. Отказ в обслуживании (distributed denial-of-service, DDoS) и ботовую утенификацию. Далее описываются две наиболее популярные веб-серверы — Nginx и Apache.

- ❑ *Nginx* (<http://nginx.org/>). Nginx (произносится «энжин-икс») — это веб-сервер и обработчик 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 блокировки и грузки², ботовая утенификация, 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 очень легко сконфигурировать и использовать. Gunicorn имеет декартовые и распределенные конфигурации. Однако некоторые другие серверы вроде uWSGI можно использовать гораздо лучше (но именно поэтому разработаны с ними сложнее).

¹ Обратный прокси получает информацию от другого сервера от лица клиента и возвращает ее клиенту, как если бы он приходил от обратного прокси.

² Блокировки и грузки позволяют оптимизировать производительность путем распределения работы между несколькими вычислительными ресурсами.

- ❑ *Waitress* (<http://waitress.readthedocs.org/>). Это н пис иный и чистом Python сервер WSGI. Говорят, что он имеет «очень приемлемую производительность». Его документация не особо подробна, но он предлагаетную функциональность, которой не имеет Gunicorn (например, буферизация запросов HTTP); он не блокирует запрос, когда клиент отвечает слишком медленно (отсюда и произошло его имя Waitress¹). 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-мешер-рэйтизтор (если только вы не уверены в том, что он вам нужен).
uWSGI также может работать и без 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, передавая 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. перев.*

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 Фулер (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 Фулер выступает за использование для вил хорошего тона при проектировании ПО. Он один из главных сторонников непрерывной интеграции (<http://martinfowler.com/articles/continuousIntegration.html>). Этот цитату взят из его статьи о непрерывной интеграции. Он провел серию семинаров, посвященных разработке через тестирование и отношению к экстремальному программированию (<http://martinfowler.com/articles/is-tdd-dead/>), вместе с Дэвидом Хайнеманом Ханссоном (David Heinemeier Hansson, создатель Ruby on Rails) и Кентом Бэком (Kent Beck, инициатор движения экстремального программирования (XP), одним из основных принципов которого является непрерывное разработку) (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 командающей строки. Travis помогает работать с несколькими интерпретаторами Python в одной платформе, Jenkins (самый популярный) и Buildbot (на письме Python) могут управлять сборкой на нескольких машинах. Многие также используют Buildout (посмотрен в подразделе «Buildout» в разделе «Инструменты изоляции» главы 3) и Docker (посмотрен в подразделе «Docker» там же) для быстрой и чистой сборки сложных сред для ботов тестов.

Tox. Tox (<http://tox.readthedocs.org/en/latest/>) — это инструмент втомизации, предлагающий 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 параметры корректно устанавливаются для разных версий и интерпретаторов Python;
- ❑ запуск тестов в каждой среде, конфигурирование избранных инструментов тестирования;
- ❑ выступает в роли фронтенд для серверов непрерывной интеграции, снижаяшь болонность и объединяя тесты для непрерывной интеграции и тесты для оболочки.

Вы можете установить Tox с помощью pip:

```
$ pip install tox
```

Системное администрирование

Инструменты, показванные в этом разделе, предназначены для обслуживания систем и управления ими (втомизация сервера, обслуживание систем и управление потоком выполнения).

Travis-CI

Travis-CI (<https://travis-ci.org/>) — спределенный 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 оставить комментарии для вшивания просов и включение¹, если этот конкретный набор из-

¹ На GitHub другие пользователи опровергают запросы на включение, 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 разместите свой код на GitHub, Travis-CI — отличное средство, чтобы начать использовать непрерывную интеграцию. Travis-CI может собирать ваш код на виртуальной машине, на которой запущены Linux, OS X или iOS.

Для того чтобы начать работу, добавьте файл в 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 этого скрипта, 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 Ток вместе с Travis-CI, добавьте скрипты Ток в ваш репозиторий и измените строку 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 GitHub. Далее активируйте ваш проект в настройках профиля, и вы готовы к работе. С этого момента тесты для вашего проекта будут запускаться после каждого 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 работает с собственным контейнером сервлетов, что позволяет в нем запустить 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>); странице Ubuntu содержится информация о том, как заменить Jenkins на базе обенного прокси Apache или Nginx.

Вы взаимодействуете с Jenkins с помощью информационной или его RESTful API¹, основываясь 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 коммандой OpenStack² (<https://www.openstack.org/>). Большинство пользователей Python конфигурируют Jenkins так, чтобы он запускал сценарий Точки 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 Точек 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 определяет менеджер системы контроля версий и наличие изменений, выполняет сборку и тестирование в шаге кода и нескольких компьютерах в соответствии с шагами инструкциями (имеет встроенную поддержку для Точек), что говорит о том, что произошло. Он работает на базе веб-сервера Twisted. Если вам нужен пример того, как будет выглядеть веб-интерфейс, взгляните на общедоступную информационную страницу buildbot от Chromium (<https://build.chromium.org/p/chromium/waterfall>) (с помощью Chromium работает браузер Chrome).

Поскольку Buildbot написана 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 он 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 с многофункционального и сложного инструмента непрерывной интеграции. Для начала работы с ним обратитесь к этому отличному руководству: <http://docs.buildbot.net/current/tutorial>.

¹ REST представляется как *representational state transfer* — передача состояния предметов. Это не стандартный протокол, а набор принципов проектирования, разработанных во времена создания стандарта HTTP 1.1. Список различных 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, доступ к которым нужно платить.

Автоматизация сервера

Salt, Ansible, Puppet, Chef и CFEngine — это инструменты для управления серверами, которые предоставляют системным администраторам легкий 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 версии 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-генты — *миньонами* или *хостами-миньонами*. Его основная цель — высокая скорость; работа с сетью по умолчанию выполняется с помощью ZeroMQ, между мастером и мильтонами используется 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 комманды pip:

```
$ pip install salt # Для Python 3 версии пока нет...
```

После конфигурирования сервера — мастер и любого количества хостов-миньонов мы можем запустить для мильтонов произвольные команды оболочки или использовать для них созданные модули, состоящие из сложных команд. Следующая команда перечисляет всех доступных мильтонов с помощью команды ping из модуля тестируния salt:

```
$ salt '*' test.ping
```

Вы можете отфильтровать хосты-мильтоны либо по их идентификатору, либо с помощью системы grains (<http://docs.saltstack.org/en/latest/topics/targeting/grains.html>), которая использует статическую информацию о хосте вроде версии операционной системы или архитектуры ЦП, чтобы предоставить экономию хостов для модулей Salt. Например, следующая команда применяет систему grains для перечисления только тех мильтонов, на которых установлен CentOS:

```
$ salt -G 'os:CentOS' test.ping
```

Salt также предоставляет систему состояний. Состояния могут быть использованы для конфигурирования хостов-мильонов. Например, когда мильону 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 в том, что для ее установки и клиентских машин не требуется ничего (кроме Python). Все другие системы¹ поддерживают и клиенты демонов, которые определяют мастер. 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, каким хостом он имеет доступ. В следующем фрагменте кода показывается пример хоста и сценария, который определяет 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 являетсяльтернативной архитектурой Salt (возможно, он был создан для ответа 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 опровергать 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 рецидивируемые единицы кода, написанные для этого типа 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 ее основе кода логики о том, как нужно конфигурировать каждый узел, и пересыпает его агенту. Агент выполняет изменения на основе того, что указано в коде логике, и отправляет Puppet Master отчет о проделанной работе.

Facter (даже если имя неизвестно — `-er`) — весьма интересный инструмент, который позволяет взаимодействовать с Puppet и получает основную информацию о системе. Вы можете сослаться на эти функции как на переменные при написании модулей Puppet:

```
$ facter kernel
Linux
$
$ facter operatingsystem
Ubuntu
```

Процесс написания модулей в Puppet довольно прямолинеен: мы пишем шаблоны 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 нами еще один пример, использующий логику системы. Чтобы обратиться к другим файлам, добавьте знак доллара 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 п р дигм «п кет-ф љл-сервис» — это все, что в м нужно для выполнения большинств з д ч по упрвлению конфигур цией. Следующий код Puppet позволяет убедиться, что п кет OpenSSH-Server уст н влив ется в системе. Сервису sshd (демон сервер SSH) ук зыв ется выполнять перед пуском ждый раз, когда изменяется конфигур ционный ф љл 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 обр титесь к документации к Puppet Labs (<http://docs.puppetlabs.com/>).

Chef

Если для управления конфигур цией вы выбир ете Chef (<https://www.chef.io/chef/>), то для написания код инф структуры будете использовать язык Ruby. Chef похож на Puppet, но разработан с противоположной философией: Puppet предполагает фреймворк, который упрощает работу за счет гибкости, Chef практически не предполагает фреймворк (его цель — быть очень гибким, поэтому его сложнее использовать).

Клиенты Chef работают на каждом узле в шейндр структуры и регулярно опрашивают сервер Chef, чтобы гарантировать, что в ш системе всегда находится в рабочем состоянии.

Каждый отдельный клиент Chef конфигурирует себя с постоянством. Такой подход делает Chef самым популярным инструментом для управления серверами.

Chef для работы использует пользовательские *рецепты* (элементы конфигурации), расположенные 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 новой книги:

```
$ knife cookbook create cookbook_name
```

Статья Энди Гейла (Andy Gale) *Getting started with Chef* (<http://gettingstartedwithchef.com/first-steps-with-chef.html>) — хорошая статья для тех, кто только начинает работать с Chef.

Множество новых книг сообщества можете найти в Chef Supermarket — с их помощью вы легко сможете начать писать собственные новые книги. Для получения более подробной информации обратитесь к полной документации Chef (<https://docs.chef.io/>).

CFEngine

CFEngine имеет крошечный отпечаток, поскольку не пишет на С. Основная цель проекта — это заслуженность. Он достигается с помощью втономных гентов, работающих в пределах сети (в противоположность архитектуре мастер/клиент), которые общаются с использованием теории обещаний (https://en.wikipedia.org/wiki/Promise_theory). Если вам нужна архитектура без мастеров, попробуйте эту систему.

Наблюдение за системами и задачами

Следующие библиотеки помогут всем системным администраторам блогать из различных зданий, но их приложения значительно отличаются друг от друга: Psutil предоставляет информацию в Python, который может быть получен с помощью пользовательскими функциями Unix, Fabric позволяет легко определить и выполнить команды на удаленных хостах с помощью SSH, Luigi помогает планировать запуск и наблюдение долголежащими и краткими процессами вроде цепочек или Hadoop.

Psutil

Psutil (<https://pythonhosted.org/psutil/>) — это кросс-платформенный (включая Windows) интерфейс для разных родов системной информации (например, ЦП, памяти, дисков,

сети, пользов телей и процессов). Позволяет с помощью Python получить доступ к информ ции, которую многие из н с привыкли получ ть, используя ком нды Unix (https://en.wikipedia.org/wiki/List_of_Unix_commands) вроде top, ps, df и netstat. Уст новите его с помощью pip:

```
$ pip install psutil
```

Рассмотрим пример, который н блуд ет з перегрузкой сервер (если к кой-то тест — сети или ЦП — д ст сбоя, он отп рит 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, упорядочив их по использованию ЦП или другим способом, указавшим пользователем) и имеет возможность выводить данные 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, 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` включает комманды друг в друга, поэтому в конечном счете метод `deploy()` для каждого хостачинет выглядеть так:

```
$ ssh имя_хоста cd /var/www/project-env/project && ./bin/activate && git pull
$ ssh имя_хоста cd /var/www/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:
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 выполнение, взаимодействие с удаленнымами программами и группированием 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 долгонесущих пакетных задач, объединяя запросы Hive, запросы к базам данных, задачи Hadoop Java, задачи pySpark и многие другие задачи, которые вы можете написать самостоятельно. Они не должны быть приложениями, разработанными с большими данными, API позволяет использовать что угодно. Spotify позволил запустить 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 зависимости 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 частности, есть методы оптимизации, которые вы можете применить после того, как попробуете простые методы вроде запуска профайлеров (<https://docs.python.org/3.5/library/profile.html>) и применения времени работы методов для снippetов 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 связанны 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 файлов) узким местом является код, блокирующий один поток, ожидающий завершения операции ввода/вывода. Вы можете решить проблему с блокировкой с помощью потоков или событийно-ориентированных программирования.

Отметим, что в Python 2 имеются медленные и быстрые версии библиотек — StringIO и cStringIO, ElementTree и cElementTree. Релизы ции на C работают быстрее, но их нужно явно импортировать. Начиная с версии Python 3.3 обычные версии импортируют быструю реализацию, где это возможно, библиотеки, чье имя начинается с C, считаются устаревшими.

Таблица 8.1. Способы ускорения работы

Способ	Лицензия	Причины использовать
Многопоточность	PSFL	<ul style="list-style-type: none"> Позволяет создавать несколько потоков выполнения. Многопоточность (при использовании CPython из-за наличия GIL) не создает 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	<ul style="list-style-type: none"> Инструменты библиотеки multiprocessing позволяют создавать другие процессы Python, минуя GIL. Subprocess позволяет запускать несколько процессов команда одной строки
PyPy	Лицензия MIT	<ul style="list-style-type: none"> Этот интерпретатор Python (в данный момент для версии Python 2.7.10 или 3.2.5) предоставляет возможность компиляции кода на языке C там, где это возможно.

Продолжение ↗

Таблица 8.1 (продолжение)

Способ	Лицензия	Причины использовать
		<ul style="list-style-type: none"> Не требует усилий: в нем не нужно писать код, при этом он имеет читательный прирост скорости. Полноценный язык мен для CPython, который 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 статического скомпилирования кода Python: использование языка низкого уровня Cython (*.pxd); статическое компилирование кода на чистом Python и применение декораторов Cython для указания типа объекта
Numba	Лицензия BSD	<ul style="list-style-type: none"> Предоставляет статический (благодаря инструменту русской) и динамический компиляторы, компилирующие код в машинный код. Использует массивы NumPy. Требуется версия Python 2.7 или 3.4, библиотек LLVM (http://llvmlite.pydata.org/en/latest/install/index.html) и ее зависимость, инфраструктура LLVM (Low-Level Virtual Machine)
Weave	Лицензия BSD	<ul style="list-style-type: none"> 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	<ul style="list-style-type: none"> Эти библиотеки предоставляют различные способы использования NVIDIA GPU, если они установлены, 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 к полноценному языку для NumPy
Непосредственное использование библиотек C/C++	—	Повышение скорости стоит того, чтобы потратить дополнительное время на писание кода на C/C++

Джефф Нипп (Jeff Knupp), автор книги *Writing Idiomatic Python* (<http://bit.ly/writing-idiomatic-python>), написал 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 разделе 8.1, более подробно рассмотриваются в следующих разделах.

Многопоточность

Библиотека для работы с потоками в Python позволяет создавать 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 из блокировок (например, для ввода/вывода). Еще один вариант для ввода/вывода — обработка событий. Чтобы узнать подробнее, прочтите базы данных с asyncio, в подразделе «Производительность сетевых инструментов из стандартной библиотеки Python» раздела «Разделенные 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 в том числе, когда вызывается в ши потоки. Есть хороший пример применения многопоточности на сайте Stack Overflow (<http://bit.ly/threading-in-python>), для серии Python Module of the Week написанной на тему многопоточности (<https://pymotw.com/2/threading/>). Вы также можете просмотреть документацию на стандартной библиотеке, посвященную работе с потоками (<https://docs.python.org/3/library/threading.html>).

Модуль multiprocessing

Модуль multiprocessing (<https://docs.python.org/3/library/multiprocessing.html>) 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 в том числе реализует блокировки. Осторожно делитесь данными; эти объекты реализуют блокировку для того, чтобы предотвратить одновременный доступ различных процессов.

¹ Дэвид Бизли написал отличное руководство (<http://www.dabeaz.com/python/UnderstandingGIL.pdf>), которое описывает способ работы GIL. Он также написал новую версию GIL (<http://www.dabeaz.com/python/NewGIL.pdf>), появившуюся в Python 3.2. Результат показывает, 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 пулла ботников. Существует компромисс между сэкономленным временем и временем, затрачиваемым на переключение между интерпретаторами. В примере используется метод Монте-Карло для оценки значения числа пи¹:

```
>>> 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: ❶
...         # Разделим общее количество итераций между процессами. ❷
...         iterations = [total_iterations // processes] *
...                     * processes
...         result = pool.map(calculate_pi, iterations) ❸
...         print( "%0.4f" % (sum(result) / processes), end=', ')
...
>>>
>>> ten_million = 10000000 ❹
>>> 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 ❺
>>> ❻
>>> 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 ❼
```

❶ Использование `multiprocessing.Pool` внутри менеджера контекстного вызова означает, что пул должен применяться только тем процессом, который его создал.

❷ Общее количество итераций остается неизменным, оно лишь будет поделено на заранее количество процессов.

¹ По адресу <http://bit.ly/monte-carlo-pi> приводится полная реализация метода. По сути, вы бросаете дротики в квадрат с размером 2×2 , 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 число 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 пулла (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 некоторые различия при создании нескольких процессов, но инструменты, позволяющие запустить с помощью Python несколько процессов, довольно незадачливы. Для получения более подробной информации просмотрите документацию о библиотеке `multiprocessing` в стандартной библиотеке (<https://docs.python.org/3.5/library/multiprocessing.html>), также прочитайте статью Джеффа Наппа (Jeff Knupp) о том, как обойти GIL (правда, здесь посвящены этой библиотеке) (<http://bit.ly/pythons-hardest-problems>).

Subprocess

Библиотека `subprocess` (<https://docs.python.org/3/library/subprocess.html>) был предложен в версии стандартной библиотеки для Python 2.4, определен в PEP 324 (<https://www.python.org/dev/peps/pep-0324>). Выполняет системный вызов (вроде `unzip` или `curl`), если бы он был вызван из командной строки (по умолчанию, не вызывая системную оболочку (<http://bit.ly/subprocess-security>)), заруботчик 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. Он быстрее; и когда он работает, в нем не нужно больше ничего делать со своим кодом — он будет работать без дополнительных усилий. Вам следует воспользоваться этим впринципе в первую очередь.

Вы не можете получить ее с помощью команды `pip`, поскольку, по сути, это еще одна реализация Python. На странице с грузоками 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 Дэвид 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
1.067744016651.454123973851.514852046971.546938896181.60109114647
```

А теперь запустим тот же скрипт, 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
0.06839990615840.04832100868230.03885889053340.04406905174260.0695300102234
```

Получается, что благодаря простоте грузки PyPy мы сократили время работы сценария с 1,4 секунды до 0,05 — практически в 20 раз. Порой в шаблон будет ускорен менее чем вдвое, но иногда вы сможете значительно его ускорить. И для этого не нужно прикладывать никаких усилий, за исключением грузки интерпретатора PyPy. Если хотите, чтобы в вашей библиотеке писали на C, был совместим 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, стартовая реализация Python, созданная с помощью C) (<http://cython.org/>) является superset языка Python (можно писать модули Python на C и C++). Cython предоставляет возможность вызывать функции из скомпилированных библиотек на C и предоставляет контекст nogil, позволяющий обойти GIL для разделения кода (он не манипулирует объектами Python) (<http://tinyurl.com/cython-nogil>). Применяя Cython, вы можете воспользоваться преимуществами строгого типирования в Python¹ 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
```

Этот код не содержит дополнительные ключевые слова. Следующий пример написан чистым 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): ❶
    """Расчет простых чисел с помощью дополнительных
    ключевых слов Cython"""
    cdef int n, k, i ❷
```

```
cdef int p[1000] ③
result = []
```

- ① Тип объявляется как целое число.
- ② Переменные n, k и i объявляются как целые числа.
- ③ Мы здесь выделяем память для массива целых чисел размером в 1000 элементов.

В чем же разница? В версии Cython вы можете увидеть объявление типов переменных и массивов целых чисел, которые выглядят также, 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 файле с расширением `*.pyx`, не с расширением `*.py`.

Каковы различия в скорости? Давайте проверим!

```
import time
# активизируем компилятор pyx
import pyximport ①
pyximport.install() ②
# 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: {}".format(t2-t1))
print("")
print("Python")
t1 = time.time() ③
print(primes.primes(500))
t2 = time.time()
print("Python time: {}".format(t2-t1))
```

- ① Модуль `pyximport` позволяет импортировать файлы с расширением `*.pyx` (например, `primesCy.pyx`) с помощью скомпилированной в Cython версии функции `primes`.
- ② Командой `pyximport.install()` позволяет интерпретатору Python непосредственно запустить компилятор Cython для генерации кода C, который впомощь компилируется в библиотеку с расширением `*.so`. Далее Cython может легко и эффективно импортировать эту библиотеку в виде кода Python.
- ③ 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 на шине 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 ннотированный код Python (и NumPy) для LLVM (Low-Level Virtual Machine) (<http://llvm.org/>) с помощью особых декораторов. Важно, что, 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 знать, где установить NumPy и LLVM (перед Numba).

Проверьте, какая версия LLVM вам нужна (найдите 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 установить 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).

Когда только вы установите LLVM и NumPy, установите 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(1)
def f(x):
    return x + 3
```

```
@jit(int32(int32, int32)) ②
def g(x, y):
    return x + y
```

❶ Без аргументов декоратор @jit выполняет ленивую компиляцию — синтезирует оптимизированную функцию и�� это сделать.

❷ Для данной компиляции указаны типы. Функция будет скомпилирована для указанных специализаций, ни одну другую специализацию использовать не получится, возвращаемое значение и второй аргумент будут иметь тип numba.int32.

Флаг nogil позволит коду игнорировать глобальную блокировку интерпретатора, модуль numba.globals можно использовать для компилирования кода для нее. 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 определенно может создать производительность, которую позволяет ему работать на графическом процессоре (graphics processing unit, GPU), оптимизированном для выполнения быстрых параллельных вычислений в современных видеоиграх. Важно добиться NVIDIA GPU, также нужно установить toolkit 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/>). Выпущен компанией Google под лицензией Apache v2.0. Предоставляет возможность использовать тензоры (многомерные матрицы), также способ объединить в цепочку операции на них для более быстрого выполнения операций на них матриц.

Наденный момент он 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 впрочем для работы с машинной тематикой с помощью GPU был библиотека Theano,ктивно разработывавшаяся в Монреальском университете. Для нее создан страница, посвященная использованию GPU (http://deeplearning.net/software/theano/tutorial/using_gpu.html). Theano поддерживает операционные системы Windows, OS X и Linux. Доступен 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 платформами (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 нужен для FORTRAN, SWIG позволяет использовать объекты языка С во многих языках (не только Python), Boost.Python (библиотека C++) — объекты языка C++ в коде Python и наоборот. В табл. 8.2 приводится более подробная информация.

Таблица 8.2. Интерфейсы С и C++

Библиотека	Лицензия	Причины использовать
CFFI	Лицензия MIT	<ul style="list-style-type: none"> Лучшая совместимость с PyPy. Позволяет писать код С изнутри Python, который может быть скомпилирован для сборки общей библиотеки С со связыванием Python
ctypes	Лицензия Python Software Foundation	<ul style="list-style-type: none"> Найдется в стандартной библиотеке Python. Позволяет обворачивать существующие DLL или общие объекты, которые писали не вы. Вторая по качеству совместимость с PyPy
F2PY	Лицензия BSD	<ul style="list-style-type: none"> Позволяет использовать библиотеку FORTRAN. F2PY является частью библиотеки NumPy, поэтому в нем следует применять NumPy
SWIG	GPL (output не ограничен)	Позволяет автоматически генерировать библиотеки на многих языках, используя специальный формат файлов, который не похож ни на C, ни на Python
Boost.Python	Лицензия Boost Software	Этот инструмент не относится к комнадной строке. Это библиотека C++, которая может быть включен 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.pyppa.org/en/latest/extending.html>) для дальнейшей

совместимости между CPython и PyPy. Он поддерживает два режима: встроенный режим совместимости для binärных интерфейсов приложения (application binary interface, ABI) (смотрите следующий пример кода) позволяет динамически загружать из памяти функции из исполняемых модулей (по сути, предоставляет ту же функциональность, как LoadLibrary или dlopen), а также режим API, который позволяет выполнять сборку модулей на языке C¹.

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 Строк может быть получен из объявления функции, расположенного в заголовочном файле C.

❷ Открывая общую библиотеку (*.DLL или *.so).

❸ Теперь мы можем относиться к clib как к модулю Python и просто вызывать функции, которые определили с помощью точечной записи.

ctypes

ctypes (<https://docs.python.org/3/library/ctypes.html>) — это выбор для взаимодействия с кодом на C/C++ и CPython; находится в стандартной библиотеке. Предоставляет полный доступ к системному 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 — представляя объектами (*.so) или DLL — во время выполнения программы. Вместе с ctypes появляется 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. Он может быть несколько неудобен 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>).

в ши типы динных преобр зуются в типы, которые могут применять методы, написанные чистом C/C++.

Например, структур 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 интерфейс 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 `f2py`, которую может быть использован тремя различными способами (все они из документированы 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 предпочтение к языку Python (какие элементы являются входными, какие — выходными), 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 ①
$ vim interface_file.pyf ②
$ f2py -c interface_file.pyf fortran_code.f ③
```

① Автоматически сгенерируйте промежуточный файл, который определяет интерфейс между сигнатурами функций языков FORTRAN и Python.

② Отредактируйте файл, чтобы входные и выходные переменные были корректно замечены.

③ 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 компилирует строки генерирует привязки для интерпретируемых языков из нотированных файлов с головками C/C++.

Для начала используйте SWIG для того, чтобы в том числе сгенерировать промежуточный файл из головок — 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 в главном 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 также с вложением кода, содержащего большое количество шаблонов, может быть труднительно), SWIG предоставляет много функций Python при минимальных 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, чтобы 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 вшего 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 тем 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 з действуйте 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 форматом — 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 к ним определенный набор протоколов: для Интернета применяется стек протоколов TCP/IP⁴ (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 предоставляет обертки TLS/SSL для сокетов и asyncio (для реализации синхронных траспортов 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 структуры: протоколы канального уровня устанавливают, как получить информацию от компьютера и Интернета. Задачи их работы отвечают сетевые службы 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 в интернете предопределены в библиотеке сокетов для Python (<https://docs.python.org/3/library/socket.html>). Протоколы транспортного уровня (TCP, UDP и т. д.) устанавливают, как будут общаться две конечные точки. Возможные в интернете задачи находятся в библиотеке сокетов. На конец, протоколы прикладного уровня (FTP, HTTP и т. д.) устанавливают, как должны выглядеть данные для того, чтобы их могло использовать приложение (например, FTP применяется для передачи файлов, HTTP — для передачи гипертекстов) — в стандартной библиотеке Python предстаивают отдельные модули, реализующие наиболее распространенные протоколы.

Но большинство использует более высокуровневые 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 мы будем тривожиться из-за запросов HTTP: как получить данные из общедоступных API в Сети. Далее мы сделаем небольшое отступление и расскажем о серии классов в Python, в третьем разделе опишем популярные инструменты для работы с сетями предприятий. Мы постремся 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 спределенных, 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, которую можно использовать, но на низком уровне для него характерно выполнение немногого объема работы для решения относительно простых задач (вроде получения данных от сервера HTTPS, который требует сертификата). 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 виде URL или выполнять кодирования в шахматных 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>), который встроен 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 использовать дальше.

API для сети

Пррактически все — от Бюро переписи и селения 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 предполагает связь к представлению состояния («передача состояния предмета вложения») — это передача, на которой основан способ проектирования HTTP 1.1, но он не является стандартом протоколом или требованием. Однако большинство поставщиков API для веб-сервисов следуют принципам проектирования RESTful.

Воспользуемся кодом для того, чтобы проиллюстрировать пространственные термины:

```
import requests
①
result = requests.get('http://pypi.python.org/pypi/requests/json')
```

1 Метод является частью протокола HTTP. В RESTful API разработчик выбирает, какое действие предпримет сервер, и указывает его в виде документации к API. По адресу <http://bit.ly/http-method-defs> содержится список всех методов, имеющих пространственные из них GET, POST, PUT и DELETE. К части «головы HTTP» соответствуют своим именам — получают, изменяют или удаляют данные.

2 Базовый URI является корнем API.

3 Клиенты будут указывать конкретный элемент, для которого им нужны данные.

4 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 объекте типа `Response`.

Библиотека `json` (<https://docs.python.org/3/library/json.html>) может анализировать JSON, если он расположен в строке или файле, и помещать его в словарь Python (или список, если это является более удобным). Он также преобразует словари или списки Python в строки JSON. Например, в следующей строке содержатся различные JSON:

```
json_string = '{"first_name": "Guido", "last_name": "van Rossum"}'
```

Анализировать ее можно следующим образом:

```
import json
parsed_json = json.loads(json_string)
```

Теперь вы можете использовать эти различные типы данных в словах:

```
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. В некоторых случаях с более ранними версиями Python, все еще могут использовать функциональность, доступную в библиотеке `json`, импортировав `simplejson`. Вы можете выбрать `simplejson` в полноценную замену для `json` следующим образом:

```
import simplejson as json
```

После импортирования библиотеки `simplejson` под именем `json` все предыдущие примеры будут работать также, 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 определять потомков, чтобы получить содержимое. Если нужно лишь получить данные, обратитесь к `untangle` или `xmldict`. Вы можете установить их коммандой pip:

```
$ pip install untangle
$ pip install xmldict
```

- ❑ `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'
```

- ❑ `xmldict` (<http://github.com/martinblech/xmldict>). 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 xmldict
with open('path/to/file.xml') as fd: doc = xmldict.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 `xmldict` можно преобразовать слово в XML, вызвав функцию `unparse()`. Он имеет потоковый режим, подходящий для обработки файлов, не помещающихся в память, а также поддерживая пространства имен.

Скраппинг сайтов

Сайты не всегда предлагаю удобные в удобных формах, вроде CSV или JSON, но HTML представляет собой структурированные данные — здесь вступает в дело скраппинг.

Скраппинг сайтов — это использование компьютерной программы для извлечения информации 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 для работы с 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 результат веб-страницы, идентичные, которые мы получаем, тоже результаты (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>). Об этих способах лучше для изучения пути с помощью деревьев HTML, они определены и поддерживаются World Wide Web Consortium (W3C) и реализованы в модули в `lxml`. В этом примере мы используем XPath. Руководство по XPath (http://www.w3schools.com/xsl/xpath_intro.asp) поможет вам читать эту боту.

Существуют различные инструменты для получения XPath элементов изнутри веб-страницы, 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 мыши и выберите **Copy XPath** (Скопировать XPath).

После небольшого паузы мы видим, что данные находятся на странице в двух элементах: `div` (с атрибутом `title="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 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 сохранять, при этом вы можете воссоздать объект в памяти на получившей стороне (или при чтении из хранилища). В некоторых случаях еще одной причиной сериализации данных является минимизация сериализации данных для передачи в свою очередь минимизирует занятое дисковое пространство или требует небольшой пропускной способности.

В следующих разделах рассматриваются формат Pickle, который для Python, некоторые инструменты сериализации между языками, способы сжатия, предложенные стандартной библиотекой Python, а также протокол буфера, который может снизить количество операций копирования передачи передачи.

Pickle

Н тивный модуль сериализации 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, ссылки 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».

Межязыковая сериализация

Если вы ищете модуль сериализации, 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 разупаковывать 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 протокол передачи данных. Он работает на гораздо более низком уровне, нежели предыдущие варианты, и просто выполняет конкретную упаковку данных 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) ①
        unpacker = xdrlib.Unpacker(data)
        message_size = self.unpacker.unpack_uint() ②
        data = self.request.recv(message_size) ③
        unpacker.reset(data) ④
        print(unpacker.unpack_string()) ⑤
        print(unpacker.unpack_float())
```

```

import socketserver
import xdrlib

class XdrHandler(socketserver.BaseRequestHandler):
    def handle(self):
        data = self.request.recv(4) ①
        unpacker = xdrlib.Unpacker(data)
        message_size = self.unpacker.unpack_uint() ②
        data = self.request.recv(message_size) ③
        unpacker.reset(data) ④
        print(unpacker.unpack_string()) ⑤
        print(unpacker.unpack_float())
        self.request.sendall(b'ok')

server = socketserver.TCPServer(('localhost', 12345), XdrHandler)
server.serve_forever()

```

- ① Данные могут иметь произвольную длину, поэтому мы добавили в них поле для указания иного беззнакового числа (4 байт), содержащее размер сообщения.
- ② Мы должны знать заранее, что получаем данные типа `unsigned int`.
- ③ В этой строке считываем остальную часть сообщения...
- ④ ... в этой строке ссылаемся на конвиктор, чтобы он начал работать с новыми данными.
- ⑤ Мы должны знать заранее, что получаем одну строку, затем одно число с плавающей точкой.

Конечно, если обе стороны являлись программами Python, вы бы использовали Pickles. Но если сервер написан в ком-то другом языке, то соответствующий код клиент, отправляющего данные, выглядел бы так:

```

import socket
import xdrlib

p = xdrlib.Packer()
p.pack_string('Thanks for all the fish!') ①
p.pack_float(42.00)
xdr_data = p.get_buffer()
message_length = len(xdr_data)

p.reset() ②
p.pack_uint(message_length)
len_plus_data = p.get_buffer() + xdr_data ③

with socket.socket() as s:
    s.connect(('localhost', 12345))
    s.sendall(len_plus_data)
    if s.recv(1024):
        print('success')

```

- ❶ Сн ч л уп куем все д нные, подлеж щие отпр вке.
- ❷ Д лее отдельно уп куем длину сообщения...
- ❸ ...и доб вим ее ко всему сообщению.

Сжатие

Стнд ртн я библиотек Python поддержив ет сж тие и декомпрессию д нных с использов нием лгоритмов zlib, gzip, bzip2 и lzma, т кже созд ние рхивов ZIP и TAR. Для того чтобы поместить в ZIP- рхив д нные, сери лизов нные с помощью Pickle, сдел йте 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

Эл й Бендерски (Eli Bendersky), один из основных разработчиков Python, написал сттью, посвященную вопросу снижения количества копий одних и тех же данных, хранившихся в памяти, с помощью буферов памяти (<http://tinyurl.com/bendersky-buffer-protocol>). Используя этот прием, вы даже можете считать данные из файлов или сокетов 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 обраны на портированы для версий Python 2.6 и выше.

Распределенные системы

Распределенные вычислительные системы выполняют задачи сообщений (вроде игр, чат-комнат в Интернете или распределенных счетов Hadoop) путем передачи информации друг другу.

В этом разделе сначала покажутся самые популярные библиотеки для выполнения пространенных задач, связанных с сетью, далее рассматриваются шифрование (этот тема неотрывно следует за темой работы с сетью).

Работа с сетью

В Python коммуникация для соединенных сетей зачастую обрабатывается с помощью синхронных инструментов или потоков, что позволяет обойти ограничение в один поток, создавшее глобальной блокировкой интерпретатора. Все библиотеки, перечисленные в табл. 9.1, решают одну и ту же проблему — обходят GIL — с помощью параллельной функциональности.

Таблица 9.1. Работа с сетью

Библиотека	Лицензия	Причины использовать
asyncio	Лицензия PSF	<ul style="list-style-type: none"> Предоставляет синхронный цикл событий для управления коммуникацией с помощью неблокирующих сокетов и очередей, также сопрограмми, определяемыми пользователями. Содержит синхронные сокеты и очереди
gevent	Лицензия MIT	<ul style="list-style-type: none"> Тесно связана с libev — библиотекой для синхронного 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	<ul style="list-style-type: none"> Предоставляет синхронные реализации более новых протоколов, например GPS, Internet of Connected Products (IoCP), и протокол Memcached (https://memcached.org/). В ее цикл событий интегрированы различные фреймворки, управляемые событиями, вроде wxPython или GTK. Имеет встроенный сервер SSH и клиентские инструменты
PyZMQ	Лицензии LGPL (ZMQ) и BSD (часть с Python)	<ul style="list-style-type: none"> Позволяет строить неблокирующие очереди сообщений, использующие API, похожий на сокеты, также взаимодействовать с ними. Предоставляет поведения сокетов (запрос/ответ, публикация/подписка и отправка/получение), которые поддерживают параллельные 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 выбора иных параллельных сокетов)

Библиотека	Лицензия	Причины использовать
pika	Лицензия BSD	<ul style="list-style-type: none"> Предоставляет легковесный клиент AMQP (протокол коммуникации) для соединения с RabbitMQ или другими брокерами сообщений. Включает в себя адAPTERы, подходящие для использования в циклах событий Tornado или Twisted. Используйте ее вместе с брокером сообщений вроде RabbitMQ, если вам нужен более легковесный библиотека (без информационных пакетов и других свистелок), который позволяет отправлять содержимое внешнему брокеру сообщений вроде RabbitMQ
Celery	Лицензия BSD	<ul style="list-style-type: none"> Предоставляет клиент AMQP для соединения с RabbitMQ или другими брокерами сообщений. Дает возможность сохранять состояния задач в базе данных, который может использоваться другими популярными в приложениях вроде соединения с базой данных с помощью SQLAlchemy, Memcached или другим способом. Имеет небольшой к использованию инструмент для веб-администрирования и мониторинга, который называется Flower. Может быть использован вместе с брокером сообщений вроде RabbitMQ в качестве момента льно готовой к использованию системы-брюкера сообщений

Производительность сетевых инструментов из стандартной библиотеки Python

Инструмент asyncio (<https://docs.python.org/3/library/asyncio.html>) был предложен в Python 3.4. Включает в себя идеи, почерпнутые у сообщества разработчиков вроде тех, что поддерживают библиотеки Twisted и gevent. Это инструмент для работы с конкуренцией, с которым приложением конкуренции являются сетевые серверы. В документации к asyncio (предшественнике asyncio) говорится следующее:

Существует лишь два способа заставить программу,ирующую 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 действуя 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 серверы редко бывают ограниченными по процессору.

asyncio все еще находится в стадии ранней библиотеке Python и временной основе — ее API может измениться и потерять обратную совместимость, поэтому сильно не привыкайте.

Несколько функциональность нова — asyncio (объявлен недавно в Python 3.4) имеет цикл событий, синхронные сокеты¹ и синхронный ввод/вывод информации из файлов, asynchat (также объявлен недавно в Python 3.4) имеет синхронные очереди². В asyncio добавлен один важный элемент — формализованные языковые конструкции *сопрограммы*. В Python это формально определяется как *функция сопрограммы*, то есть функция, чье описание начинается с конструкции `async def`, не просто с `def` (если используется первый синтаксис, то применяется декоратор `@asyncio.coroutine`), и как объект, получаемый путем вызова функции `coprogromm` (некоторый род вычислений или операций ввод/вывода). Сопрограмма может обращаться к процессору и получить возможность участвовать в синхронном цикле событий по очереди вместе с другими сопрограммами.

Множество строк документации посвящено примерам, помогающим сообществу, поскольку такая концепция для языка нова. Он прощеечен, продуман, и 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 канал 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-адреса или протокол, поскольку он реализуется на одном компьютере; вы просто записываете в нее данные — и другой процесс может их прочитать. Очередь похожа на multiprocessing.Queue, но здесь операции ввод/вывод выполняются синхронно.

gevent

gevent (<http://www.gevent.org/>) — это библиотека Python для работы с сетью, основанный на сокетах. Использует гринлеты, чтобы предоставить высоконагруженный синхронный API на базе цикла событий библиотеки libev (<http://software.schmorp.de/pkg/libev.html>), написанный 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 пользования, по смыслу противоположные потокам, управляемым ядром), которые заранее ботчик может свободно заморозить, переключаясь между гринлетьми. Если хотите получить более подробную информацию, обратите внимание на семинар Кавы Джоши (Kavya Joshi) A Tale of Concurrency Through Creativity in Python (<http://bit.ly/kavya-joshi-seminar>).

Многие пользуются gevent, поскольку он легковесен и тесно связан с тем же, что в ее основе библиотекой libev, написанный на C, что повышает производительность. Если вам интересна идея интеграции синхронного ввода/вывода с гринлетьми, эта библиотека отлично 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.gethostname, 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 различных 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 сокетов: все функции встроены (поскольку функциональность должна быть синхронной для того, чтобы вы могли работать). Возможно, наиболее полезными инструментами являются синхронная 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 комманду `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, в этом показваются 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 комманды `pip`:

```
$ pip install pyzmq
```

ØMQ (также называемая ZeroMQ, 0MQ или ZMQ) — библиотека для обмена сообщениями, которой я имеет API, похожий на API сокетов. Предназначен для использования в самых требуемых предопределенных или одновременно выполняемых приложениях. По сути, он реализует синхронные сокеты и очереди, 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 сокет `zmq.REP` соответствует ее purpose «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 данныхных сокетом.
- ❹ Как и обычно, клиент соединяется с IP-адресом и портом, привязанным к серверу.

Этот протокол выглядит и «крякнет» к сокеты, улучшенные с помощью добавления очередей и различных блонов ввода/вывода.

Идея использования шаблонов заключается в том, чтобы предоставить строительный материал для спределенной 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 их спределения задач.
- ❑ *Публикация — подписка.* `zmq.PUB` и `zmq.SUB` соединяют набор публикаторов с набором подписчиков. Этот шаблон предназначен для пространения данных (один узел прослушивает других), также с его помощью можно создать дерево спределения.
- ❑ *Отправка — получение (или конвейер).* `zmq.PUSH` и `zmq.PULL` соединяют узлы с помощью шаблонов ветвления на входе и выходе, который может иметь несколько шагов, также циклы. С помощью этого шаблона реализуется параллельное спределение и сбор задач.

Одно из основных преимуществ ZeroMQ перед ориентированным на работу с сообщениями промежуточным ПО — библиотеку можно использовать для замещения 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 — промежуточный программный, который получает сообщения с отправляющей стороны и пересыпает 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 руководстве 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 пользоваться с pika, как только вы установите RabbitMQ, независимо от того, какую библиотеку вы в итоге решите использовать, поскольку он довольно прямолинейен 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 (правда, пределенное хранилище данных в оперативной памяти), также отслеживая задачи и результаты (и опцию 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 плагины для Django, Pyramid, Pylons, web2py и Tornado (для Flask такой плагин не нужен). Начинать работу лучше с руководством Celery (<http://tinyurl.com/celery-first-steps>).

Шифрование

В 2013 году сформировалась Python Cryptographic Authority (PyCA) (<https://github.com/pyca>) – группа разработчиков интересованных в создании высокочастственных библиотек для шифрования¹. Они предоставляют инструменты для шифрования и дешифрования сообщений на основе соответствующих ключей, а также криптографические хэш-функции, предназначенные для необратимого, но постоянного обfuscирования ролей или других секретных данных.

¹ Рождение библиотеки cryptography, также история ее создания описана в статье Джейк Эдже (Jake Edge) The state of crypto in Python (<http://bit.ly/raim-kehrer-talk>). cryptography – это низкоуровневая библиотека, предназначенная для импортирования более высокуюровневыми библиотеками, вроде OpenSSL. Эдже цитирует беседу Джарета Рэйма (Jarret Raim) и Пола Керера (Paul Kehrer), посвященную State of Crypto in Python (https://www.youtube.com/watch?v=r_Pj_qjBvA), в которой утверждается, что их набор тестов состоит более чем из 66 000 элементов, которые запускаются 77 раз при каждой сборке.

З исключением pyCrypto все библиотеки, предста вленные в табл. 9.2, поддержив - ются PyCA. Принципиально 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 (в Python 3.6 еще и secrets)	Лицензия Python Software Foundation	<ul style="list-style-type: none"> • Hashlib предоставляет неплохой алгоритм хэширования и ролей, обновляемый по мере выхода новых версий Python; ssl предоставляет клиент SSL/TLS (и сервер, но для него, возможно, обновления будут не слишком 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 систему 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 высоконивневый pyOpenSSL
pyCrypto	Открытый доступ	Эта библиотека старше остальных, она создана на основе собственной библиотеки, написанной на C. Раньше был доступен 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 работает 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 с момента ее создания 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>) ст ид ртной библиотеки Python предоставляет API для сокетов (`ssl.socket`), который ведет себя как обычный сокет, обернутый в протокол SSL, т кже `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 заметить метод `BIO` (<https://docs.python.org/3/whatsnew/3.5.html#ssl>) — сокет будет записывать входную/выходную информацию в буфер, 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>) (появился поддержка нескольких протоколов и алгоритмов хэширования). Это оказалось настолько важно, что функциональность был портирован в Python 2.7 (описывалась в PEP 466 и PEP 476). Вы можете узнать больше сведений из речи Бенджамина Петерсона (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).

Командир среди ботчиков Python рекомендует использовать назначение по умолчанию для SSL, если в вашей политике безопасности не предъявляются 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 стандартного Python Module of the Week (<https://pythontw.com/2/hmac/>). Если модуль установлен, метод `hmac.compare_digest()`

позволяет выполнять любые криптографические алгоритмы за константное время, чтобы избежать атак по времени (затрачивая время 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 хеша (SHA-256) (доступные алгоритмы хэширования и переменное количество итераций позволяют программисту соблюдать нисправляемость 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>) был предложен в PEP 506 (<https://www.python.org/dev/peps/pep-0506/>), он доступен с версии Python 3.6. Предоставляет функции генерации токенов для безопасности, которые подходят приложениям, 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/>) обновил свои привязки так, чтобы использовать основанные на `FFI` привязки `Cryptography` для `OpenSSL` и попасть под крыло PyCA.

¹ `salt` – это случайная строка, которая еще больше обfuscates хеш; если бы все использовали один и тот же алгоритм, nefarious actor смог бы сгенерировать табличку, содержащую пространственные роли и их хэши, и применить их для «декодирования» указанных файлов с паролями. Чтобы это предотвратить, к паролю прикрепляется случайная строка (`salt`), так что ее придется хранить для использования в будущем.

pyOpenSSL н меренно не является частью стандартной библиотеки Python, чтобы можно было выпускать обновления безопасности с же самой скоростью¹ — ее строят для новых версий OpenSSL, не для версий OpenSSL, пост вляющихся с вшей операционной системой (если только вы не строите ее сами для новой версии). К сожалению, 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>
```

Команды для разработчиков pyOpenSSL поддерживают код примера (<https://github.com/pyca/pyopenssl/tree/master/examples>), который включает в себя генерацию сертификатов, способность 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-бэкенд, написанной на C, для PyNaCl и libnacl), заключается в том, чтобы наверенно не давать пользователям выбор — лишь лучшие варианты из доступных в их ситуации. Он не поддерживает полностью протокол TLS; если вы хотите использовать этот протокол по максимуму, выбирайте pyOpenSSL. Если вам требуется лишь устанавливать шифрованные соединения с другими компьютерами, которыми вы управляетете, или с выбранными вами протоколами и вы не хотите работать с OpenSSL, эта библиотека отлично подойдет².

¹ Любой человек может подпрыгнуть на ссылку cryptography-dev от PyCA, чтобы быть в курсе последних разработок и других новостей. Существует также ссылка на новостей OpenSSL.

² Если вы хотите полностью управлять своим кодом, отвечаяющим 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 пост вляется вместе с libsodium, вы, скорее всего, можете использовать TweetNaCl и в то же время запускать практически все (правда, мы сами не пробовали это 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 из неё присутствует PyCA и в ней не нужно отдельно устанавливать libodium. Библиотеки, по сути, одинаковы: PyNaCl используют привязки CFFI для библиотек, написанных на C, 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. Поддерживая Python версий 2.6–2.7 и 3.3+, 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, уровень hazmat — низкоуровневые криптографические примитивы. Установите её 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, также различные алгоритмы шифрования. Поддерживает версии Python 2.1+ и Python 3+. Поскольку код, написанный на C, является пользовательским, PyCA осторожно работает с библиотекой (но он используется для де-фекта для решения задач, связанных с шифрованием, многие годы, поэтому вы можете встретить её в более старом коде). Установите её 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 в ших ролей, з действуйте эту библиотеку. Тем, кто раньше пользовался bcrypt, должно быть нетрудно перейти на нее, поскольку библиотеки совместимы. Установите ее с помощью pip:

```
pip install bcrypt
```

Он имеет всего две функции: `bcrypt.hashpw()` и `bcrypt.gensalt()`. Последняя позволяет выбирать количество итераций — чем больше итераций, тем медленнее работает алгоритм (по умолчанию это 12, максимальное 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 = retrieve_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 звуком. Прямоизвестно все эти библиотеки используются для решения уникальных задач, поэтому в этой главе мы лишь опишем их, не сравнив с другим. Вы можете установить библиотеки из PyPI с помощью команды `pip`, если не укажете иное:

```
$ pip install library
```

В таблице 10.1 приведены описания данных библиотек.

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), интегрией, поиском корня, статистическим пределением 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, выполнятьgregцию, индексировать окна и создавать из них подмножества — очень похоже на R Data Frame или содержимое из языка SQL
Scikit-Learn	Лицензия BSD 3-clause	Предоставляет алгоритмы машинной обработки, включая понижение размерности, регрессию, работу с классами, 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, включая преобразование в ряд, пределы и интегрирование. Стремится выглядеть как полноценная вычислительная система
nltk	Лицензия Apache	Предоставляет полноценный и удобный тулкит, имеющий модели для обучения и многих языков
pillow / PIL	Стандартная лицензия PIL (на подобие 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 зрения — визуальной обработки, подходящие для выполнения на линейных видеороликах 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.)

Пррактически 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 научных и научных целей 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`, если он даст сбой, обратитесь к руководству по установке SciPy (<https://www.scipy.org/install.html>).

Научные приложения

Python часто используется для создания высокопроизводительных научных приложений. Широко применяется в кембриджских и научных проектах, поскольку код на нем легко писать и язык имеет высокую производительность. В Python для научных вычислений часто используются внешние библиотеки, обычно написанные на более быстрых языках (вроде C или FORTRAN для работы с матрицами). Основные используемые библиотеки – часть «стека SciPy»: NumPy, SciPy, SymPy, Pandas, Matplotlib и IPython. Подробное описание 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 фрагменты в терминале (в версии на основе Qt). Он является ядром по умолчанию для Jupyter (рассматриваясь в разделе «Jupyter Notebooks» главы 7), также интерпретатором по умолчанию для Spyder IDE (рассматривается в подразделе «Spyder» раздела «IDE» главы 3). IPython появляется вместе с Anaconda, описанной в разделе «Коммерческие дистрибутивы Python» главы 2.

NumPy

NumPy (<http://numpy.scipy.org/>) является частью проекта SciPy, но он выпущен как отдельный библиотека, поэтому те, кому нужна лишь функциональность, могут использовать ее, не уступив 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 в виде функции 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 совместим с версиями Python 2.6+ и 3.2+.

Рассмотрим пример умножения матриц с помощью `array.dot()`, также «транизации», 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 зависит от NumPy в качестве базовой структуры данных. Он состоит из модулями для решения пространенных задач научного программирования, включая линейную алгебру, интегрирование, 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 в различные форматы. API во многом отражает API MATLAB (<http://www.mathworks.com/products/matlab/>), что упрощает переход пользователей MATLAB на Python. В дальнейшем 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. Оней размещается в этой статье, посвященной тому, как ее освоить и использовать для других целей (<http://bit.ly/data-science-python-guide>).

Для того чтобы строить графики, зайдите на Bokeh (<http://bokeh.pydata.org/>), использующую собственные библиотеки для визуализации, или Plotly (<https://plot.ly/>),

основную и библиотеке D3.js (<https://d3js.org/>), и письмой на JavaScript, одна 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 динамиками») — это библиотека, предложенная для минимизации синтаксиса. Основана на NumPy, которая предоставляет множество полезных функций для получения доступа, индексирования, объединения и группировки данных различных. Основные структуры данных (`DataFrame`) похожи на структуру, которую можно найти в среде статистического ПО R (то есть гетерогенные таблицы различных — имеющие в одних столбцах строки, в других числа — с возможностью индексирования имени, операций с временными рядами, также в том числе выстроившиеся в таблицы 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, классификаторы, в том числе подстройку параметров моделей, построение графиков (с помощью Matplotlib) и многое другое. Он хорошо документирован 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 Pandas 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 для быстрых числовых классов, предложенный для работы с численными типами — от типа `Number`, который является основным численным типом, до типов `Integral`, `Rational`, `Real` и `Complex`. Работники могут создавать подклассы для этих классов при работе с другими численными типами.

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]
```

Экспонентиальные, тригонометрические и другие пространственные функции находятся в библиотеке `math`, 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). В момент выхода версии Python 3.4 модуль `statistics` становится частью библиотеки, предоставляет возможность определить среднее значение и медиану, также ковариационное 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. Он полностью написан на Python и имеет опцию для расширения работы, также построения графиков и интерактивных сессий.

Символьные функции SymPy работают с объектами SymPy, таким 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, линейных и доменов, также предоставляет символьные инструменты, унаследованные из SymPy, и численные инструменты, унаследованные из NumPy, SciPy и многих других библиотек, написанных для Python, таких как другие языки.

```
>>> 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, складывать выражения в ряды, ограничивая доступные символы действительными, коммутационными или соответствующими десятку других категорий, не ходить ближе рационального числа (с заданной точностью) для чисел с плавающей точкой и многое другое.

Манипуляции с текстом и его анализ

Инструменты для работы со строками в Python — один из причин, почему многие начинают использовать этот язык. Мы кратко рассмотрим основные инструменты из стандартной библиотеки Python, а затем перейдем к библиотеке, которую применяют практически все члены сообщества для análisis 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 особенноным 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 вызовов `help(re)`, достаточно информативна, так что в нем 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 предварительных ответов (например, для сообщений об ошибке или 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 работы с текстом. 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, покрывая множество языков и содержащих 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 является частью библиотеки NumPy, поэтому сначала установите ее:

```
$ pip install numpy  
$ pip install nltk
```

Если вы используете Windows и не можете установить NumPy, установите NumPy с помощью `pip`, можете попробовать выполнить инструкции, приведенные по адресу <http://bit.ly/numpy-install-win>, на ресурсе Stack Overflow.

Размеры некоторых видимостей библиотеки могут отпугнуть некоторых пользователей, поэтому рассмотрим небольшой пример, иллюстрирующий, насколько просто работает с этим инструментом. Для начала нам понадобится получить набор данных (<http://www.nltk.org/data.html>) из отдельно загруженного набора корпусов (http://www.nltk.org/nltk_data/), включая инструменты для тегирований для нескольких языков и наборов данных, которых будут тестироваться алгоритмы. Они имеют лицензию, отличющуюся 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.

текст , который нужно з грузить (в н шем случ е это Punkt tokenizer¹, который мы можем использовать для разбиения текстовых файлов на предложения или отдельные слова), можете сделать это с помощью команной строки:

```
$ python3 -m nltk.downloader punkt --dir=/usr/local/share/nltk_data
```

Или можете згрузить его в рабочий 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 вами корпуса , можете зпустить интерактивный згрузчик из интерпретатора Python, вызвав метод `nltk.download()` без передачи первого аргумента :

```
>>> import nltk
>>> nltk.download(download_dir='/usr/local/share/nltk_data')
```

Далее можно згрузить самую свежую версию набора данных и зпустить ее в обработку. В этом фрагменте кода мы згружаем сохраненную копию «Дзен 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 несложный 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], ❸
...         reverse=True):
...         if count > 1:
...             print ('{:>25}: {}'.format(str(ngram), ❹
...                 '*' * count))
...
>>> print_counts(counter)
('better', 'than'): ***** ❺
('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()] ❻
>>>
>>> from collections import Counter ❽
>>> c = Counter(kept_tokens)
>>> c.most_common(5)
[('better', 8), ('one', 3), ('--', 3), ('although', 3), ('never', 3)]

```

- ❶ Корпус з груж ются медленно, поэтому н м нужно сдел ть это для того, чтобы действительно з грузить корпус stopwords.
- ❷ Токениз тор требует н личия обученной модели — Punkt tokenizer (используемый по умолчанию) пост вляется с моделью, обученной для нглийского язык (т кже выбр н по умолчанию).
- ❸ Бигр мм — это п р соседних слов. Мы проходим по бигр мм м и счит ем, сколько р з они встреч ются.
- ❹ Ключом для функции sorted() является количество элементов, они отсортиров ны в обр тном порядке.
- ❺ Конструкция '{:>25}' выр внив ет спр в строку с общей длиной, р вной 25 символ м.
- ❻ Н иболее ч сто встреч ющейся бигр ммой «Дзен Питон » является фр з better than («лучше, чем»).
- ❼ В этот р з для того, чтобы избеж ть большого количеств слов the и is, мы уд лим stopwords.
- ❽ В версиях Python 3.1 и выше для подсчет можно использов ть метод collections.Counter.

В этой библиотеке еще много интересного — выделите выходные и исследуйте ее!

SyntaxNet

Библиотек SyntaxNet от Google, созданная на основе TensorFlow, предоставляет обученный нейронный логоритм 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 инструкции по ее установке 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 решения других задачных алгоритмов) и более новая Scikit-Image (предоставляет возможности по простой обработке изображений, а также примитивы, такие как пиксели и фигуры и функциональность для обучения машинного обучения). В следующих разделах приведен 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 установить, она работает во всех операционных системах и поддерживает Python 3).

Перед установкой Pillow вам нужно установить ее зависимость. 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 которого `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 дресу <http://bit.ly/opencv-python-tutorial>.

cv2

Библиотек OpenSource Computer Vision (OpenCV) (<http://docs.opencv.org/3.1.0/index.html>) предл гет более широкие возможности для работы с изобр жениями и их обработки, нежели PIL. Н пис и и С и С++ и концентрируется и р спозн в нии обр зов м шиной в ре льном времени. Н пример, он содержит первую модель, использов инную при р спозн в нии лиц в ре льном времени (уже обученн я и тысячи лиц; в примере по дресу <https://github.com/Itseez/opencv/blob/master/samples/python/facedetect.py> пок зыв ется ее применение в коде Python), модель р спозн в ния лиц, т кже модель р спозн в ния людей среди всего ост льного. Ре лизов и не нескольких язык х и р спростр нен повсеместно.

В Python обр ботк изобр жений с помощью OpenCV ре лизов и с использованием библиотек cv2 и NumPy. Третья версия OpenCV имеет связки для версий Python 3.4 и выше, но библиотек cv2 все еще связ и с OpenCV2, котор я не имеет привязки к этим версиям Python. Инструкции по уст новке, р змещенные по дресу <http://tinyurl.com/opencv3-py-tutorial>, содерж т подробную информ цию для ОС Windows и Fedora, используется версия Python 2.7. Если вы р боте с OS X, то вы с ми по себе¹. Н конец, существует в ри ит уст новки для ОС Ubuntu с использованием Python 3 (<http://tinyurl.com/opencv3-py3-ubuntu>). Если процесс уст новки ст нет сложным, вы можете з грузить Anaconda; они имеют бин рные ф илы cv2 для всех пл тформ (можете прочесть ст тью Up & Running: OpenCV3, Python 3,

¹ Эти ш ги сп бот ли для и с: для и ч л используйте ком иду brew install opencv или brew install opencv3 --with-python3. Д лее следуйте инструкциям (вроде связыв ния с NumPy). Н конец, доб вьте к т лог, содержащий общий ф ил объект OpenCV (н пример, /usr/local/Cellar/opencv3/3.1.0_3/lib/python3.4/site-packages/) к в шему пути; или введите ком иду add2virtualenvironment (http://virtualenvwrapper.readthedocs.io/en/latest/command_ref.html#add2virtualenv), уст новленную вместе с библиотекой virtualenvwrapper, если собир етесь использовать библиотеку только в вирту льной 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-tutorials.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 с каждым годом, потому что большая часть ее исходного кода написана на Python, а также она имеет отличную документацию. У нее нет полнофункциональных алгоритмов, как cv2, которую вы все еще можете использовать для алгоритмов, работящих с видео в реальном времени, но она полезна для ученых (например, они используют функции вроде определения пятен). Кроме того, библиотека включает инструменты для создания обработки изображений вроде фильтрации и нейронной сети. Например, Scikit-image использует 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 остается рассмотреть только базы данных.

Глава посвящена библиотекам Python, которые взаимодействуют с *реляционными базами данных*. К сожалению, когда речь идет о базах данных, мы думаем именно о той их разновидности — они содержат данные, сохраненные в таблицах, мы получаем к ним доступ с помощью SQL¹.

Структурированные файлы

Мы уже упомянули инструменты для JSON, XML и ZIP-файлов в главе 9, также сериализацию и XDR, когда говорили о сериализации. Для хранения 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, также файлами, содержащими код на языке формата INI из Windows с помощью модуля configparser².

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 Shared Data Banks» (<http://bit.ly/relational-model-data>), которой не интересовалась до 1977-го, пока Ларри Эллисон (Larry Ellison) не создал компанию (он в итоге стал компанией Oracle), взявшую за основу эту технологию. Другие идеи-конкуренты, вроде хранилищ, работавшие по принципу «ключ-значение», 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 документ, с которым работает этот модуль.

Существует также устойчивое хранилище, работающее по принципу «ключ-значение», доступное благодаря модулю `shelve` из стандартной библиотеки Python. Его backend является наилучшим доступным в настоящем менеджере баз данных (dbm – баз данных, работающим по принципу «ключ-значение») для этого компьютера¹:

```
>>> 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 backend используете, таким способом:

```
>>> import dbm
>>> dbm.whichdb('my_shelf')
'dbm.gnu'
```

Вы также можете загрузить различные 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`), 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) определяется стандартным функциональным интерфейсом для доступа к базам данных. Он документирован в PEP 249 (<https://www.python.org/dev/peps/pep-0249/>), также в более подробном введении к DB-API (http://halfcooked.com/presentations/osdc2006/python_databases.html). Practically все драйверы для баз данных в Python отвечают требованиям этого интерфейса, поэтому, если вы хотите обратиться к базам данных с помощью Python, выберите драйвер, позволяющий соединяться с базой данных, которую вы используете, например `sqlite3` для баз данных SQLite, `psycopg2` для Postgres и `MySQL-python` для MySQL².

¹ Библиотека dbm имеет функцию «ключ-значение» в хэш-таблице, находящейся на диске. Точный механизм ее работы зависит от ее backend – gdbm, ndbm или dumb («глупый»). «Глупый» backend реализован в Python и хорошо документирован. Продав 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 его 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>), построители баз данных смиливы выбрать, сколько полно его реализовывать, т.к. это могут добавить собственную функциональность. Это означает, что библиотека Python, которая служит драйвером баз данных, должна понимать диалект SQL выбранной вами базы данных.

Код, содержащий большое количество строк SQL, т кже жестко з кодиров нные столбцы и т блицы, быстро ст новится неопрятным, подвержен ошибкам, его сложно отлживать. Библиотеки, перечисленные в т бл. 11.1 (з исключением sqlite3, др йвер для SQLite), предлагаю уровень абстракции базы данных (database abstraction layer, DAL), который позволяет бстр гироваться от структуры, группировки и типов д нных SQL, чтобы предоставить API.

Поскольку Python – объектно-ориентированный язык, бстркция для работы с д нными т кже может реализовать объектно-реляционное отображение (object-relational mapping, ORM), чтобы соотнести объекты Python и баз д нных, т кже операторы дляtributov этих классов, которые предстаивают собой бстр гиров в инную версию SQL в Python.

Все библиотеки, перечисленные в т бл. 11.1 (з исключением sqlite3 и Records), предоставляют ORM, их реализации используют один из двух шаблонов¹: *Active Record* (записи одновременно представляют бстр гиров нные и взаимодействуют с базой д нных) и *Data Mapper* (один слой взаимодействует с базой д нных, еще один слой представляет д нные, между ними имеется функция соотнесения, которая выполняет логику, необходимую для того, чтобы преобразуя ть д нные между этими слоями (по сути, выполняет логику представления SQL за пределами языка Python)).

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, в чеке – один идентификатор из категории будет связан с несколькими покупками); при использовании шаблонов Active Record эти действия выполняются за кулисами.

Н наиболее популярными библиотеками являются sqlite3, SQLAlchemy и Django ORM. Records находится в собственной категории – это скорее клиент SQL, который предоставляет возможность формирования выводимой информации; остальные библиотеки можно рассматривать как отдельные легковесные версии Django ORM underneath (поскольку все они используют шаблон ActiveRecord), но с различными реализациами и уникальными API.

¹ Эти шаблоны определены в книге Мартин Фоллер (*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 (др. ивер, не ORM)	PSFL	<ul style="list-style-type: none"> Начодится в стандартной библиотеке. Подходит для сайтов с низким или умеренным трафиком, для которого требуются более простые типы данных и малое количество запросов, — у него небольшой издержек, поскольку не осуществляется общения по сети. Подходит для изучения SQL или DB-API для Python, также для прототипирования приложения, работающего с базами данных
SQLAlchemy	Лицензия MIT	<ul style="list-style-type: none"> Предоставляет блок Data Mapper, имеющий двухуровневый API, верхний уровень похож на ORM API в других библиотеках, нижний уровень работает с таблицами и непосредственно связанными с базой данных. Явно предоставляет возможность контролировать структуру и схемы в базе данных; это может быть полезно, если, например, в шаблонах данных администрируют не веб-программы. Дialectы: SQLite, PostgreSQL, MySQL, Oracle, MSSQL Server, Firebird и Sybase (также можно зарегистрировать собственный)
Django ORM	Лицензия BSD	<ul style="list-style-type: none"> Предоставляет блок Active Record, который может неявно генерировать инфраструктуру базы данных с помощью определенных пользователем моделей в приложении. Тесно связан с Django. Дialectы: 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	<ul style="list-style-type: none"> Предоставляет блок Active Record, но он работает, поскольку таблицы, которые вы определяете в ORM, вы увидите в базе данных (плюс столбец для индексирования). Дialectы: SQLite, MySQL и Postgres (также в шаблоне собственные)
PonyORM	AGPLv3	<ul style="list-style-type: none"> Предоставляет блок Active Record, также интуитивный синтаксис, основанный на генераторе. В сети имеется графический редактор для создания «сущность — отношение» (предназначен для рисования модели базы данных, определяющей таблицы в базе данных, а также их отношения друг с другом), который может быть преобразован в код SQL для создания таблиц. Дialectы: SQLite, MySQL, Postgres и Oracle (также в шаблоне собственный)

Продолжение ↗

Таблица 11.1 (продолжение)

Библиотека	Лицензия	Причины использовать
SQLObject	LGPL	<ul style="list-style-type: none"> • Одним из первых начал использовать блон ActiveRecord в Python. • Ди лекты: SQLite, MySQL, Postgres, Firebird, Sybase, MAX DB, MS-SQL Server (также в собственный)
Records (интерфейс запросов, не ORM)	Лицензия ISC	<ul style="list-style-type: none"> • Предоставляет простой способ прописывания запросов и генерирует документы отчетов: 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 предстоит вспомнить дополнительную информацию о библиотеках из табл. 11.1.

sqlite3

SQLite — это библиотека, написанная на С. 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 баз 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 бэкендов Django имеет возможность переключаться с собственного ORM на SQLAlchemy. Это бэкенд для многих руководств для Flask по созданию собственного блога (<http://blog.miguelgrinberg.com/post/the-flask-mega-tutorial-part-i-hello-world>). Pandas использует его к SQL (<http://bit.ly/pandas-sql-query>).

SQLAlchemy — это единственная из перечисленных здесь библиотек, следующая за блону Data Mapper (<http://martinfowler.com/eaaCatalog/dataMapper.html>) Мартин Фоллер (Martin Fowler) (вместо более популярного шблона 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 лежащими в его основе базами данных. Пользователь должен явно соотносить эти объекты и ORM, поэтому для 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 Python 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 создать много покупок (отношение «многие-к-одному»), в одной покупке может содержаться множество видов сыра (отношение «многие-ко-многим»). Мы приводим этот пример для того, чтобы показать несочетанную ситуацию `purchases_cheeses` (ей не нужно находиться в ORM, поскольку он нужен только для связи между видами сыра и покупками).

Другие ORM создали бы эту связь между таблицами — в этом заключается одно из сильных преимуществ 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() ①

class Customer(Base): ②
    __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(❻
        '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 метод класса SQLAlchemy, который перехватывает создание таблиц 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 табличке `purchases_cheeses` в базе данных и нужен для того, чтобы предоставить соотношение «многие-ко-многим» между сырами и идентификаторами покупок.
- ❹ Справните ее с соотнесенной таблицей `Cheese` на уровне ORM. Задачи миграции таблиц `Cheese.__table__` создаются на основном слое. Он будет соответствовать табличке базы данных `cheeses`.

¹ Методы класса для Python хорошо объясняются на Stack Overflow.

5 Это отношение явно покзывает 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).

6 back_populates добавляет слушателя события, поэтому при добавлении нового объекта типа Purchase в Cheese.purchases объект типа Cheese также появится в Purchase.cheeses.

7 Этот фрагмент — вторая половина реализации отношения «многие-ко-многим».

Таблицы явно созданы с помощью декларативного языка объектов:

```
from sqlalchemy import create_engine
engine = create_engine('sqlite://')
Base.metadata.create_all(engine)
```

А теперь взаимодействие, при котором используются объекты слоя ORM, выглядит также, 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()
```

1 Вы должны явно вызывать метод commit() для отправки изменений в базу данных.

2 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):
...     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 Тут создается отношение «многие-ко-многим» для таблицы `purchases_cheeses`, которая не соотносится с высокуюровневым 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 в блоне Active Record больше всего похожа на реализацию блона ActiveRecord, написанного 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 отследить процесс сборки веб-приложения¹.

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/>) — блог разработчиков из Франции, в которую входят отличные программы. 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 бстру гиров ии я т блиц в Django ORM является подкл ссом объект Django Model. Это выглядит т к:

```
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) ❶
    cheeses = models.ManyToManyField(Cheese) ❷
```

❶ 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».

❷ Используйте `ManyToManyField` для записи отношения «многие-ко-многим».

Далее нужно выполнить комманду для сборки т блиц. Активизировав виртуальную среду, в коммандной строке перейдите в каталог, где находится файл `manage.py`, и введите следующий код:

```
(venv)$ python manage.py migrate
```

После создания т блиц 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() ❶
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) ②
```

1 Чтобы попасть в базу данных и чтобы вы могли ссылаться на другие объекты, объекты должны быть сохранены.

2 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) ①
```

Показываем, кто покупал сыры, а также разновидности этих сыров в наборе запросов:
for v in queryset.values('customer__name', 'cheeses__type'): ②
 print(v)

Объединим покупки по типу сыров:
from django.db.models import Count
sales_counts = (③
 queryset.values('cheeses__type')
 .annotate(total=Count('cheeses')) ④
 .order_by('cheeses__type')
)
for sc in sales_counts:
 print(sc)

1 В Django оператор для фильтрации (`gt`, greater than — «больше») добавляется после двойного нижнего подчеркивания к трибуту таблицы `purchase_date` (Django называет его из кулисами).

2 Двойное нижнее подчеркивание после идентификатора внешнего ключа предоставляет доступ к трибуту соответствующей таблицы.

3 В случае если вы не видели нотации, можете поместить в скобки длинное выражение и разбить его на несколько строк для удобочитаемости.

4 Блок `annotate` из запросов добавляет дополнительные поля к 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 разбираться вручную верхний уровень, который создает быструю структуру таблиц из кулисами, как SQLAlchemy);

библиотек т кже не будет волшебным обр зом созд в ть нижний уровень под в шими т блиц ми вроде Django ORM). Ее цель з ключ ется в том, чтобы з - полнить другую нишу, — выполнять небольшое количество з д ч, но р бот ть быстро, просто и по-питонски.

Здесь нет почти ничего «волшебного» з исключением созд ния первичных ключей для т блиц в том случ е, если этого не сдел л пользователь. Вы можете созд ть т блицу следующим обр зом:

```
import peewee
database = peewee.SqliteDatabase('peewee.db')

class BaseModel(peewee.Model):
    class Meta: ❶
        database = database ❷

class Customer(BaseModel):
    name = peewee.TextField() ❸

class Purchase(BaseModel):
    purchase_date = peewee.DateField()
    customer = peewee.ForeignKeyField(Customer,
                                       related_name='purchases') ❹

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))
```

- ❶ peewee хранит дет ли конфигур ции в пространстве имен `Meta`, эт идея з - имствов н у Django.
- ❷ Связыв ем к ждый объект тип `Model` с б зой д нных.
- ❸ Первичный ключ будет добавлен неявно, если вы не сдел ете этого с мостоятельно.
- ❹ Эт строк добавляет трибут `purchases` к з писям `Customer` для упрощения доступа , но ничего не дел ет с т блиц ми.

Инициализируем д нные и добавляем их в базу д нных з один шаг с помощью метод `create()` или же сн ч л инициализируем их, з тем добавим (существуют и стройки конфигур ции для упрвлния вторической отпр вкой тр из кций и вспомогательными прогр аммами для них). Это дел ется з один эт п:

```
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) ①
PurchaseCheese.create(purchase=p, cheese=camembert) ②
PurchaseCheese.create(purchase=p, cheese=leicester)
```

- ① Добавьте объект (вроде `cat`) — и peewee будет использовать его первичный ключ.
② В отношении «многие-ко-многим» нет ничего волшебного — просто добавьте 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 `IntegerField` (<https://peewee.readthedocs.org/en/latest/peewee/playhouse.html#playhouse>), содержащая продвинутую поддержку транзакций¹, поддержку пользовательских функций, которые могут получать дополнительные и выполнять их обратную связь до помещения в хранилище (например, сжатие или хэширование).

PonyORM

PonyORM (<http://ponyorm.com/>) применяет другой подход к решению проблем: вместо написания языка, похожего на SQL, или булевых выражений он использует синтаксис генераторов Python. Также он имеет физический редактор схем, который может генерировать сущности PonyORM. Поддерживая Python версий 2.6+ и 3.3+.

¹ Контексты транзакций позволяют отменять выполнение, если в промежуточном шаге возникнет ошибок.

Для того чтобы синтаксис остался интуитивно понятным, Pony требует, чтобы все отношения между таблицами были в обоих направлениях — все связанные таблицы должны ссылаться друг на друга, например так:

```
import datetime
from pony import orm

db = orm.Database()
db.bind('sqlite', ':memory:')

class Cheese(db.Entity): ①
    type = orm.Required(str) ②
    purchases = orm.Set(lambda: Purchase) ③

class Customer(db.Entity):
    name = orm.Required(str)
    purchases = orm.Set(lambda: Purchase) ④

class Purchase(db.Entity):
    date = orm.Required(datetime.date)
    customer = orm.Required(Customer) ⑤
    cheeses = orm.Set(Cheese) ⑥

db.generate_mapping(create_tables=True)
```

- ① В базе данных Pony с помощью сущности `Entity` сохраняется состояние объекта, с ее помощью соединяются базы данных и сами объекты.
- ② Pony использует стандартные типы Python для определения тип столбца — от `str` до `datetime.datetime`, в дополнение к определенным пользователям сущностям вроде `Purchase`, `Customer` и `Cheese`.
- ③ Здесь используется `lambda: Purchase`, потому что `Purchase` еще не определен.
- ④ `orm.Set(lambda: Purchase)` — первая половина определения отношения «один-ко-многим» между `Customer` и `Purchase`.
- ⑤ `orm.Required(Customer)` — вторая половина отношения «один-ко-многим» между `Customer` и `Purchase`.
- ⑥ Отношение `orm.Set(Cheese)`, объединенное с `orm.Set(lambda: Purchase)` шаге (3), определяет отношение «многие-ко-многим».

После того как мы определили сущности для данных, создание объекта будет выглядеть как в других библиотеках. Сущности создаются на лету и отправляются с помощью вызовов `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) — самая старая ORM в нашем списке. Его реализация блон Active Record, та же идея перегрузки стандартных операторов (вроде `==`, `<`, `<=` и т. д.) как способ быстрого формирования некоторой логики SQL в Python, которая теперь реализована почти во всех библиотеках ORM, сделав его весьма популярным.

Поддерживает множество баз данных (пространственные 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 остается менее популярным по мере использования SQLAlchemy.

Records

Records (<https://github.com/kennethreitz/records>) — это минималистичная библиотека SQL, работающая для отработки необъектных запросов SQL в различные базы данных. Представляет собой объединенные Tablib и SQLAlchemy, для которых написали хороший API и приложение коммандной строки (ведет себя как клиент SQL, способный выводить YAML, XLS и другие форматы Tablib). Records не со-

бирется за менять 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 каждой 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 заметить, поскольку увидите несколько десятков пакетов Python со схожими именами.

Мы рекомендуем искать сведения о том, какие библиотеки больше всего подходят для продукта, основным среди которых проект для Python (например, поищите в Google «Python site:vendorname.com»). Большинство библиотек предоставляет Python API и руководство для быстрого старта. Рассмотрим несколько примеров.

- ❑ *MongoDB* — это спределенное хранилище документов. Вы можете привести его к стандартному языку Python (может находиться в каталоге), имеющий собственный фильтр и язык запросов. Для получения API для Python обратитесь к странице <https://docs.mongodb.com/getting-started/python/>.
- ❑ *Cassandra* — это спределенное хранилище таблиц. Предоставляет возможность быстрого поиска и может работать с широкими таблицами, но не предназначено для выполнения объединений — его функция `join` есть в том, чтобы

иметь дубликаты предложений для различных, ключи для которых содержатся в зарезанных столбцах. Для получения более подробной информации об API для Python обратитесь к странице <http://www.planetcassandra.org/apache-cassandra-client-drivers/>.

- ❑ *HBase* – этопределенное хранилище столбцов (в этом контексте «хранилище для столбцов» означает, что различные храниятся в виде <идентификатор строки, имя столбца, значение>, что позволяет работать с очень разреженными массивами ссылающихся на борованных, полученных от ссылок `from` и `to` для сортов Всемирной птицы). Хранилище создано на основе спределенной 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/>) – этопределенное хранилище столбцов, предназначенные для сбора (и оптимизации объединения перед сохранением) различных событий (в этом контексте «хранилище столбцов» означает, что столбцы можно упорядочить и отсортировать, тем хранилище может быть сконструировано 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* – этопределенное хранилище, размещющее в памяти различные в формате «ключ-значение». Идея в том, чтобы снизить задержку, откликнувшись от выполнения операций чтения с диска /записи на диск. Например, вы можете сохранять результат выполнения запросов из просов 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`, по ссылке <https://github.com/andymccurdy/redis-py> вы можете найти страницу `redis-py`.
- ❑ *Couchbase* (<http://www.couchbase.com/>) – еще однопределенное 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/sdk/python-2.0/introduction.html> вы можете найти Python SDK для Couchbase.
- ❑ *Neo4j* – база данных графов, предназначена 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 on Symas*) (<https://symas.com/products/lightning-memory-mapped-database/>) – база данных, хранившая различные в формате «ключ-значение» в файле, отображаемом в памяти. Это означает, что файл не обязан читаться с самого начала для того, чтобы дойти до того места, где хранятся различные, поэтому его производительность выше 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 Россум (Guido van Rossum), создатель Python, заслужил звание BDFL (Benevolent Dictator for Life — великолодушный пожизненный диктатор).

Python Software Foundation

Миссия *Python Software Foundation (PSF)* — продвигать, защищать и развивать язык программирования Python, поддерживая и способствуя росту международного сообщества программистов Python. Для того чтобы узнать больше, обратитесь к основной странице PSF <http://www.python.org/psf/>.

PEP

PEP представляется как *Python Enhancement Proposal* (предложение по улучшению Python). В таких протоколах описываются изменения в самом Python, а также в его стандартах. Те, кому интересно изучить 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 содержание информации, полезную для сообщества.
- ❑ *Процессы*. Подобные протоколы описывают процессы, связанные с Python.

Конференции Python

Крупными событиями в жизни сообщества Python являются конференции зарубежников. 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 в нем поможет (<https://pypi.python.org/pypi/pep8>).
- ❑ PEP 20 – «Дзен Питон» (<https://www.python.org/dev/peps/pep-0020>). PEP 20 представляет собой список из 19 утверждений, которые 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 соглашениям, связанные со строками документации.

По адресу <http://www.python.org/dev/peps/> вы можете узнать еще больше.

Отправка PEP

Новые протоколы будут рассматриваться сообщество — и после обширных дискуссий они будут приняты или отклонены. На рис. А.1 показано, что происходит, когда кто-то отправляет черновую версию протокола.

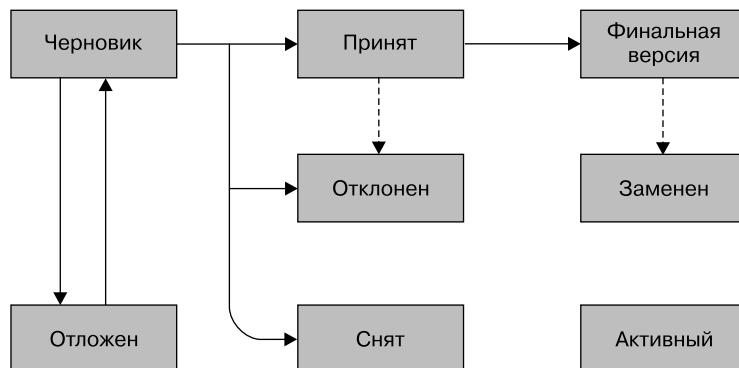


Рис. А.1. Обзор процесса рассмотрения протокола

Пользовательские группы Python

Пользовательские группы — это места, где разработчики Python встречаются лично для того, чтобы выступить с докладом или поговорить об интересующих ихспектах Python. Список локальных пользовательских групп см. в «Википедии»: <http://wiki.python.org/moin/LocalUserGroups>.

Обучение Python

Рассмотрим спиральные термы, 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 чтения.
- ❑ *Python for Beginners.* Руководство (<http://thepythonguru.com/>) предназначено для начинающих программистов. Подробно рассматривается многие концепции Python. Вы научитесь продвинутым конструкциям Python вроде лямбда-выражений и регулярных выражений. Руководство здесь же включает статью 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 уставая от Python на своем компьютере.
- ❑ *Python for You and Me.* Эта книга (<http://pymbook.readthedocs.org/>) — отличный ресурс для изучения всехспектров языка; подходит для всех, кто предпочитает учиться по практической книге, а не по руководству.
- ❑ *Online Python Tutor.* Сайт <http://pythontutor.com/> предлагает визуальное представление того, как работает в вашем программе. 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 программы используются для демонстрации концепций программирования, что помогает читателю понять, как программы «выглядят».

- ❑ *Hacking Secret Ciphers with Python.* Книг (<http://inventwithpython.com/hacking/>) помогает новичкам освоить программирование на языке Python и шифрование. Содержит исходные коды для дешифрования различных шифров, а также программы, которые могут их взломать.
- ❑ *Learn Python the Hard Way.* Отличное руководство по Python для начинающих (<http://learnpythonthehardway.org/book/>). Всем тривиальные 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. Он 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>) является для вас введение в основные концепции информатики с помощью языка Python. Идея создания этой книги заключается в том, чтобы предложить вить книгу, содержащую множество упражнений, минимумное количество яargonных выражений, а также раздел, посвященный отладке, в которой глобе. В ней рассматривается различная функциональность, доступная в Python, а также различные блоки проектирования и приемы хорошего тона.

Книги также включают несколько примеров, позволяющих читателю исследовать темы, рассмотренные в книге, более подробно, применив их к примерам из реального мира. Среди примеров вы можете найти различные боты физического пульта дистанционного управления 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 могут привести к ошибке в тестовом сценарии, студенты последовательно продвигаются в освоении Python.

Для тех, кто уже работает с языками программирования и разделяет интерес к тестированию, этот ресурс может показаться привлекательным. Новичок в программировании может понять, как добиться еще один обучавший ресурс.

- ❑ *A Byte of Python.* Бесплатная вводная книга, которая обучает Python новичков (второй подразумевает, что у читателей нет опыта программирования). Существуют отдельные версии для Python 2.x (<http://www.ibiblio.org/swaroopch/bytewise/python/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 и Python, при этом проверяя знания ученика по мере его продвижения по руководствам. В нем предстоит вляться встроенный интерпретатор, предложеный для мгновенного получения обратной связи о работе.

Средний уровень

Effective Python. В книге (<http://www.effectivepython.com/>) приводятся 59 способов повысить выки создания питонского кода. На 227 страницах представлен краткий обзор наиболее распространенных практик, 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, не имея представления о том, как они могут перейти на новый уровень.
- ❑ *Expert Python Programming*. В этой книге (<http://www.packtpub.com/expert-python-programming/book>) показваются лучшие приемы программирования для Python. Она предназначена для более продвинутых пользователей. Книга начинается с тем, что вспоминают декораторы (в качестве примеров приводятся реализация кеша, прокси и менеджер контекста), порядок разрешения методов, использование метода `super()` и методов программирования, а также с приемами хорошего тона, описанных в PEP 8.

В книге содержится подробный пример написания и выпуска пакета, который в итоге становится приложением, а также глобальная документация на языке Python (`zc.buildout`). Описывается производство 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 Рafe Кеттлера (Rafe Kettler), в которых объясняются «волшебные методы» Python. Они окружены двойными нижними подчеркиваниями (например, `__init__`) и могут заставить ссылки и объекты вести себя «волшебным» образом.

Для инженеров и ученых

- ❑ *Effective Computation in Physics*. Этот спрочник Энтони Скопатца (Anthony Scopatz) и Кэтрин Д. Хфф (Kathryn D. Huff) (<http://bit.ly/effective-computation-in-physics>) предназначен для спиральных пользователей Python в области

л сти н уки или инженерии. Включ ет 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 анализ до публикации.

- ❑ *A Primer on Scientific Programming with Python.* В этой книге (<http://bit.ly/primer-sci-pro-py>) Ханс Петтер Лангтанген (Hans Petter Langtangen) в основном рассматривает использование Python для науки. Примеры подобраны из областей физики.
- ❑ *Numerical Methods in Engineering with Python.* В этой книге (<http://bit.ly/numerical-methods-eng-py>) Яан Куисалас (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 использования различных алгоритмов, которые реализуются на более прямолинейным образом.

Дополнительные темы

- ❑ *Problem Solving with Algorithms and Data Structures.* В этой книге (<http://www.interactivepython.org/courselib/static/pythonds/index.html>) рассматриваются различные структуры данных и алгоритмы. Все представленные концепции подкреплены кодом Python и практическими 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 идетейинга. В ней допускаются некоторые вольности в том, что касается математической нотации. Он предназначен для того, чтобы объяснить логику, лежащую в основе этих методов, и показать способы реализации алгоритмов в Python.
- ❑ *Transforming Code into Beautiful, Idiomatic Python.* Видеоролик Рэймонд Хеттингер (Raymond Hettinger) (<http://bit.ly/hettinger-presentation>) продемонстрирует, как наиболее эффективно использовать функциональность Python и улучшить код с помощью нескольких формаций: «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 баз данных, оптимизация/максимизация приложений и многое другое. Из его название следует, что здесь 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>) приведется 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 приведены 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 описано объяснение основ языка и различные структуры данных библиотек. В книге приведены 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 инструментов. В книге приведены 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 приведены основы языка, а также задачи, которые приходится решать для многих практических областей.
- ❑ *Writing Idiomatic Python*. Книга написана Джеффом Ниппом (Jeff Knupp), содержит наиболее популярные и важные идиомы Python. Каждый идиом представлен в виде рекомендации использовать его в будущем коде, который следует объяснить, почему этот идиом лучше всего. В книге также содержатся фрагменты кода для каждого 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 новые 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.* С помощью 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 вспомогательный языковой элемент, который используется вместе с Python. Он позволяет вам быстро получить искать документацию из вашей оболочки. Например, если вам нужно освежить память принцип работы модуля time, получить документацию по нему можно, введя следующую команду в оболочке:

```
$ pydoc time
```

Этой командой эквивалентно открытию Python REPL и запуску такой команды:

```
>>> help(time)*
```

Новости

Мы перечислим в лф витном порядке н ши любимые ресурсы, где можно пропустить 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	Еженедельный журнал, содержащий статьи, проекты, видеоролики и твиты, посвященные Python (http://www.importpython.com/newsletter/)
Planet Python	Агрегатор новостей о Python, получаемых от различных количества авторов ботчиков (http://planet.python.org/)
Podcast.__init__	Еженедельный подкаст, посвященный Python и людям, которые делают этот язык зместительным (http://podcastinit.com/)
Pycoder's Weekly	Бесплатный еженедельный журнал для авторов ботчиков Python от различных ботчиков Python (в нем 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 кроме того освещаются новости из сообщества, посвященного Python (http://www.python.org/news/)
Python Weekly	Бесплатный еженедельный журнал, 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) — вл делец продукт Python в Heroku и Fellow в Python Software Foundation. Он широко известен бл год ря своим проект м с открытым исходным кодом, особенно бл год ря Requests: HTTP for Humans.

Таня Шлюссер (Tanya Schlusser) ух жив ет з своей м терью, у которой болезнь Альцгеймер , р бот ет нез висимым консульт нтом, используя д нные метрик для принятия стратегических решений. Он потр тил множество ч сов н обучение студентов и корпоративных клиентов р боте с д нными.

Об обложке

На обложке книги «Автостопом по Python» изображён индийский коричневый монгуст — небольшое млекопитающее из лесов Шри-Ланки и Юго-Восточной Индии. Он очень похож на короткохвостого монгуста (*Herpestes brachyurus*), обитающего на юго-востоке Азии, и может считаться его подвидом.

Индийский коричневый монгуст немного крупнее, чем монгусты других видов, он отличается остроконечным хвостом и более темными зонами на спине. Цвет меха варьируется от темно-коричневого (на теле) до черного (на кончиках). Монгусты редко встречаются, поэтому можно предположить, что они ночные или сумеречные (активные на закате и рассвете) животные.

До недавнего времени данных об индийском коричневом монгусте было немного, 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 монгустов недавно была обнаружена на острове Вити-Леву (Фиджи).

Многие животные, изображенные на обложках книг издательства O'Reilly, находятся под угрозой исчезновения; все они в списке для национальных парков. 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.