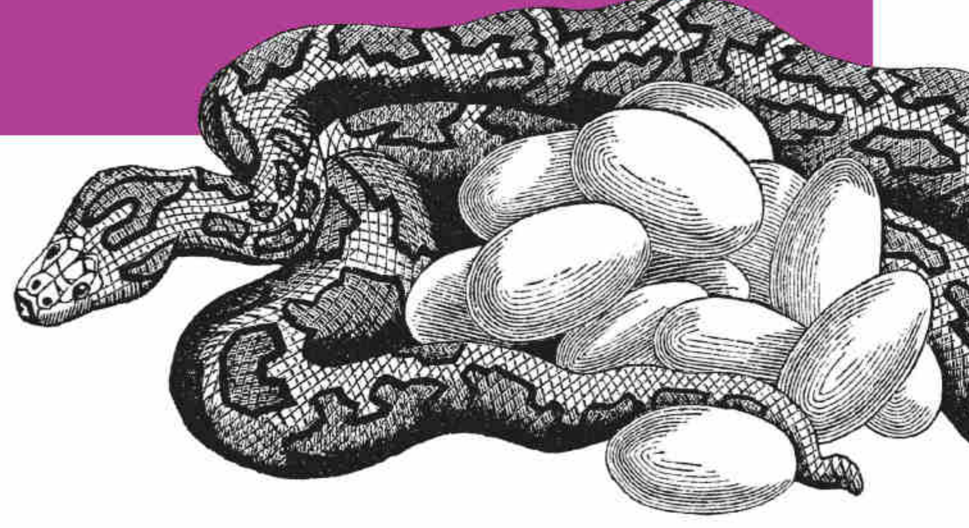
O'REILLY

Простой Python

Современный стиль программирования



Билл Любанович

Introducing Python

Bill Lubanovic

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

Билл Любанович

Простой Python

Современный стиль программирования

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

2016

ББК 32.973.2-018.1
УДК 004.43
Л93

Любанович Билл

Л93 Простой Python. Современный стиль программирования. — СПб.: Питер, 2016. — 480 с.: ил. — (Серия «Бестселлеры O'Reilly»).

ISBN 978-5-496-02088-6

Эта книга идеально подходит как для начинающих программистов, так и для тех, кто только собирается осваивать Python, но уже имеет опыт программирования на других языках. В ней подробно рассматриваются самые современные пакеты и библиотеки Python. Стилистически издание напоминает руководство с вкраплениями кода, подробно объясняя различные концепции Python 3. Под обложкой вы найдете обширный материал от самых основ языка до сравнительно сложных и узких тем.

Прочитав эту книгу, вы не только убедитесь, что Python — это вкусно, но и освоите искусство тестирования, отладки, многократного использования кода, а также научитесь применять Python в различных предметных областях.

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

ББК 32.973.2-018.1
УДК 004.43

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.

ISBN 978-1449359362 англ.
ISBN 978-5-496-02088-6

© Copyright c 2015 Bill Lubanovic. All rights reserved
© Перевод на русский язык ООО Издательство «Питер», 2016
© Издание на русском языке, оформление ООО Издательство «Питер», 2016
© Серия «Бестселлеры O'Reilly», 2016

Краткое содержание

Введение 22

Об авторе 27

Глава 1. Python: с чем его едят 28

Глава 2. Ингредиенты Python: числа, строки и переменные 43

Глава 3. Наполнение Python: списки, кортежи, словари и множества 70

Глава 4. Корочка Python: структуры кода 100

Глава 5. Py Boxes: модули, пакеты и программы 142

Глава 6. Ой-ой-ой: объекты и классы 157

Глава 7. Работаем с данными профессионально 180

Глава 8. Данные должны куда-то попадать 210

Глава 9. Распутываем Всемирную паутину 257

Глава 10. Системы 281

Глава 11. Конкуренция и сети 302

Глава 12. Быть питонщиком 346

Приложения

Приложение А. Пи-Арт 382

Приложение Б. За работой 395

Приложение В. Ру в науке 408

Приложение Г. Установка Python 3 428

Приложение Д. Ответы к упражнениям 437

Приложение Е. Вспомогательные материалы 473

Оглавление

Введение 22
Аудитория 22
Краткое описание 22
Версии Python 24
Соглашения, принятые в этой книге 25
Использование примеров кода 25
Как с нами связаться 26
Благодарности 26

Об авторе 27

Глава 1. Python: с чем его едят 28
Python в реальном мире 33
Python против языка X 33
Почему же Python? 37
Когда не стоит использовать Python 37
Python 2 против Python 3 38
Установка Python 39
Запуск Python 39
Интерактивный интерпретатор 39
Файлы Python 40
Что дальше? 41
Момент просветления 41
Упражнения 42

Глава 2. Ингредиенты Python: числа, строки и переменные 43
Переменные, имена и объекты 43
Числа 47
Целые числа 47
Приоритет операций 51
Системы счисления 52
Преобразования типов 53
Насколько объемен тип int? 55
Числа с плавающей точкой 55
Математические функции 56
Строки 56
Создаем строки с помощью кавычек 57
Преобразование типов данных с помощью функции str() 59
Создаем управляющие символы с помощью символа \ 60
Объединяем строки с помощью символа + ... 61
Размножаем строки с помощью символа * 61
Извлекаем символ с помощью символов [ ] 61
Извлекаем подстроки с помощью оператора [start : end : step] 62
Получаем длину строки с помощью функции len() 65
Разделяем строку с помощью функции split() 65
Объединяем строки с помощью функции join() 66
Развлекаемся со строками 66
Регистр и выравнивание 67
Заменяем символы с помощью функции replace() 68
Больше действий со строками 69
Упражнения 69

Глава 3. Наполнение Python: списки, кортежи, словари и множества 70
Списки и кортежи 70
Списки 71
Создание списков с помощью оператора [] или метода list() 71
Преобразование других типов данных в списки с помощью функции list() 71

Получение элемента с помощью конструкции [смещение] 72
Списки списков 73
Изменение элемента с помощью конструкции [смещение] 74
Отрежьте кусочек — извлечение элементов с помощью диапазона смещений 74
Добавление элемента в конец списка с помощью метода append() . . . 75
Объединяем списки с помощью метода extend() или оператора += 75
Добавление элемента с помощью функции insert() 76
Удаление заданного элемента с помощью функции del 76
Удаление элемента по значению с помощью функции remove() 77
Получение заданного элемента и его удаление с помощью функции pop() 77
Определение смещения элемента по значению с помощью функции index() 77
Проверка на наличие элемента в списке с помощью оператора in . . . 78
Определяем количество включений значения с помощью функции count() 78
Преобразование списка в строку с помощью функции join() 78
Меняем порядок элементов с помощью функции sort() 79
Получение длины списка с помощью функции len() 80
Присваивание с помощью оператора =, копирование с помощью функции copy() 80
Кортежи 81
Создание кортежей с помощью оператора () 82
Кортежи против списков 83
Словари 83
Создание словаря с помощью {} 84
Преобразование с помощью функции dict() 84
Добавление или изменение элемента с помощью конструкции [ключ] 85
Объединение словарей с помощью функции update() 87
Удаление элементов по их ключу с помощью del 87
Удаление всех элементов с помощью функции clear() 88
Проверяем на наличие ключа с помощью in 88

Получение элемента словаря с помощью конструкции [ключ] 89
Получение всех ключей с помощью функции keys() 89
Получение всех значений с помощью функции values() 90
Получение всех пар «ключ — значение» с помощью функции items() 90
Присваиваем значения с помощью оператора =, копируем их с помощью функции copy() 90
Множества 91
Создание множества с помощью функции set() 92
Преобразование других типов данных с помощью функции set() 92
Проверяем на наличие значения с помощью ключевого слова in 93
Комбинации и операторы 94
Сравнение структур данных 97
Создание крупных структур данных 97
Упражнения 98

Глава 4. Корочка Python: структуры кода 100
Комментируем с помощью символа # 100
Продлеваем строки с помощью символа \ 101
Сравниваем выражения с помощью операторов if, elif и else 102
Повторяем действия с помощью while 106
Прерываем цикл с помощью break 107
Пропускаем итерации с помощью continue 107
Проверяем, завершился ли цикл заранее, с помощью else 108
Выполняем итерации с помощью for 108
Прерываем цикл с помощью break 110
Пропускаем итерации с помощью continue 110
Проверяем, завершился ли цикл заранее, с помощью else 110
Итерирование по нескольким последовательностям с помощью функции zip() 111
Генерирование числовых последовательностей с помощью функции range() 112
Прочие итераторы 113

Включения 113
Включение списков 113
Включение словаря 116
Включение множества 116
Включение генератора 117
Функции 118
Позиционные аргументы 122
Аргументы — ключевые слова 122
Указываем значение параметра по умолчанию 122
Получаем позиционные аргументы с помощью * 124
Получение аргументов — ключевых слов с помощью ** 125
Строки документации 125
Функции — это объекты первого класса 126
Внутренние функции 128
Замыкания 129
Анонимные функции: функция lambda() 130
Генераторы 131
Декораторы 132
Пространства имен и область определения 134
Обработка ошибок с помощью try и except 137
Создание собственных исключений 139
Упражнения 140

Глава 5. Py Boxes: модули, пакеты и программы 142
Отдельные программы 142
Аргументы командной строки 143
Модули и оператор import 143
Импортируем модуль 143
Импортируем модуль с другим именем 145
Импортируем только самое необходимое 145
Каталоги поиска модулей 146
Пакеты 146

Стандартная библиотека Python . . . 147
Обработка отсутствующих ключей с помощью функций setdefault() и defaultdict() . . . 148
Подсчитываем элементы с помощью функции Counter() . . . 150
Упорядочиваем по ключу с помощью OrderedDict() . . . 151
Стек + очередь == deque . . . 152
Итерируем по структурам кода с помощью itertools . . . 153
Выводим данные на экран красиво с помощью функции pprint() . . . 155
Нужно больше кода . . . 155
Упражнения . . . 156

Глава 6. Ой-ой-ой: объекты и классы . . . 157
Что такое объекты . . . 157
Определяем класс с помощью ключевого слова class . . . 158
Наследование . . . 160
Перегрузка метода . . . 161
Добавление метода . . . 162
Просим помощи у предка с помощью ключевого слова super . . . 163
В защиту self . . . 164
Получаем и устанавливаем значение атрибутов с помощью свойств . . . 165
Искажение имен для безопасности . . . 168
Типы методов . . . 169
Утиная типизация . . . 170
Особые методы . . . 172
Композиция . . . 175
Когда лучше использовать классы и объекты, а когда — модули . . . 176
Упражнения . . . 178

Глава 7. Работаем с данными профессионально . . . 180
Текстовые строки . . . 180
Unicode . . . 180
Формат . . . 188
Совпадение с регулярными выражениями . . . 192

Бинарные данные . . . 200
bytes и bytearray . . . 200
Преобразуем бинарные данные с помощью модуля struct. . . 202
Другие инструменты для работы с бинарными данными. . . 205
Преобразование байтов/строк с помощью функции binascii() . . . 206
Битовые операторы. . . 206
Упражнения. . . 207

Глава 8. Данные должны куда-то попадать. . . 210
Ввод информации в файлы и ее вывод из них . . . 210
Запись в текстовый файл с помощью функции write() . . . 211
Считываем данные из текстового файла с помощью функций read(), readline() и readlines() . . . 213
Записываем данные в бинарный файл с помощью функции write(). . . 215
Читаем бинарные файлы с помощью функции read() . . . 216
Закрываем файлы автоматически с помощью ключевого слова with. . . 216
Меняем позицию с помощью функции seek() . . . 216
Структурированные текстовые файлы . . . 218
CSV. . . 219
XML. . . 221
HTML . . . 223
JSON. . . 223
YAML. . . 226
Безопасность. . . 228
Конфигурационные файлы . . . 228
Другие форматы обмена данными . . . 229
Сериализация с помощью pickle. . . 230
Структурированные бинарные файлы . . . 231
Электронные таблицы . . . 231
HDF5. . . 231
Реляционные базы данных. . . 232
SQL. . . 233
DB-API. . . 234

SQLite . . . 234
MySQL . . . 236
PostgreSQL . . . 237
SQLAlchemy . . . 237
Хранилища данных NoSQL . . . 244
Семейство dbm . . . 244
Memcached . . . 245
Redis . . . 246
Прочие серверы NoSQL . . . 254
Full-Text Databases . . . 255
Упражнения . . . 255

Глава 9. Распутываем Всемирную паутину . . . 257
Веб-клиенты . . . 258
Тестируем с telnet . . . 259
Стандартные веб-библиотеки Python . . . 260
За пределами стандартной библиотеки: requests . . . 262
Веб-серверы . . . 263
Простейший веб-сервер Python . . . 263
Web Server Gateway Interface . . . 265
Фреймворки . . . 265
Bottle . . . 266
Flask . . . 268
Веб-серверы, не использующие Python . . . 272
Другие фреймворки . . . 274
Веб-сервисы и автоматизация . . . 276
Модуль webbrowser . . . 276
API для Сети и Representational State Transfer . . . 277
JSON . . . 278
Поиск и выборка данных . . . 278
Получаем HTML-код с помощью BeautifulSoup . . . 278
Упражнения . . . 280

Глава 10. Системы . . . 281
Файлы . . . 281
Создаем файл с помощью функции open() . . . 281
Проверяем существование файла с помощью функции exists() . . . 282
Проверяем тип с помощью функции isfile() . . . 282
Копируем файлы с помощью функции copy() . . . 283
Изменяем имена файлов с помощью функции rename() . . . 283
Создаем ссылки с помощью link() или symlink() . . . 283
Изменяем разрешения с помощью функции chmod() . . . 284
Изменение владельца файла с помощью функции chown() . . . 284
Получаем pathname с помощью функции abspath() . . . 285
Получаем символьную ссылку с помощью функции realpath() . . . 285
Удаляем файл с помощью функции remove() . . . 285
Каталоги . . . 285
Создаем каталог с помощью функции mkdir() . . . 285
Удаляем каталог с помощью функции rmdir() . . . 286
Выводим на экран содержимое каталога с помощью функции listdir() . . . 286
Изменяем текущий каталог с помощью функции chdir() . . . 287
Перечисляем совпадающие файлы с помощью функции glob() . . . 287
Программы и процессы . . . 287
Создаем процесс с помощью модуля subprocess . . . 288
Создаем процесс с помощью модуля multiprocessing . . . 290
Убиваем процесс с помощью функции terminate() . . . 290
Календари и часы . . . 291
Модуль datetime . . . 292
Модуль time . . . 295
Читаем и записываем дату и время . . . 297
Альтернативные модули . . . 300
Упражнения . . . 300

Глава 11. Конкуренция и сети 302
Конкуренция 303
Очереди 304
Процессы 305
Потоки 306
Зеленые потоки и gevent 308
twisted 311
asyncio 312
Redis 313
Помимо очередей 316
Сети 317
Шаблоны 317
Модель публикации-подписки 318
TCP/IP 322
Сокеты 323
ZeroMQ 327
Scapy 331
Интернет-службы 332
Веб-службы и API 334
Удаленная обработка 335
Большие данные и MapReduce 340
Работаем в облаках 341
Упражнения 344

Глава 12. Быть питонщиком 346
О программировании 346
Ищем код на Python 347
Установка пакетов 348
Используем pip 348
Менеджер пакетов 349
Установка из исходного кода 349
Интегрированные среды разработки 349
IDLE 350
PyCharm 350
IPython 350

Именуйте и документируйте . . . 351
Тестируем код . . . 352
pylint, pyflakes и PEP-8 . . . 352
unittest . . . 354
Пакет doctest . . . 358
Пакет nose . . . 359
Другие фреймворки для тестирования . . . 360
Постоянная интеграция . . . 361
Отлаживаем свой код . . . 361
Отлаживаем с помощью pdb . . . 362
Записываем в журнал сообщения об ошибках . . . 368
Оптимизируем код . . . 371
Измеряем время . . . 371
Алгоритмы и структуры данных . . . 373
Cython, NumPy и расширения C . . . 374
PyPy . . . 375
Управление исходным кодом . . . 375
Mercurial . . . 375
Git . . . 376
Клонируйте эту книгу . . . 378
Как узнать больше . . . 378
Книги . . . 379
Сайты . . . 379
Группы . . . 380
Конференции . . . 380
Coming Attractions . . . 380

Приложения

Приложение А. Пи-Арт . . . 382
2D-графика . . . 382
Стандартная библиотека . . . 382
PIL и Pillow . . . 383
ImageMagick . . . 386
Графические пользовательские интерфейсы (Graphical User Interface, GUI) . . . 386

Трехмерная графика и анимация 388
Диаграммы, графики и визуализация 391
matplotlib. 391
bokeh 392
Игры 393
Аудио и музыка 393

Приложение Б. За работой 395
The Microsoft Office Suite 395
Выполняем бизнес-задачи 397
Обработка бизнес-данных 397
Извлечение, преобразование и загрузка 398
Дополнительные источники информации. 401
Python в области финансов 402
Безопасность бизнес-данных 402
Карты 403
Форматы 403
Нарисуем карту. 404
Приложения и данные. 407

Приложение В. Ру в науке 408
Математика и статистика в стандартной библиотеке 408
Математические функции 408
Работа с комплексными числами 410
Рассчитываем точное значение чисел с плавающей точкой с помощью decimal 411
Выполняем вычисления для рациональных чисел с помощью модуля fractions 412
Используем Packed Sequences с помощью array 412
Обработка простой статистики с помощью модуля statistics 413
Перемножение матриц 413
Python для науки 413
NumPy 414
Создание массива с помощью функции array() 414
Создание массива с помощью функции arange() 415

Создание массива с помощью функций zeros(), ones() и random() . . . 416
Изменяем форму массива с помощью метода reshape() 417
Получаем элемент с помощью конструкции [] 418
Математика массивов 419
Линейная алгебра 420
Библиотека SciPy 421
Библиотека SciKit 421
Библиотека IPython 421
Лучший интерпретатор 422
Блокноты IPython 423
Pandas 426
Python и научные области 427

Приложение Г. Установка Python 3 428
Установка стандартной версии Python 428
Mac OS X 431
Windows 432
Linux или Unix 432
Установка Anaconda 432
Установка и использование pip и virtualenv 435
Установка и использование conda 436

Приложение Д. Ответы к упражнениям 437
Глава 1. Python: с чем его едят 437
Глава 2. Ингредиенты Python: числа, строки и переменные 438
Глава 3. Наполнение Python: списки, кортежи, словари и множества . . . 438
Глава 4. Корочка Python: структуры кода 442
Глава 5. Py Boxes: модули, пакеты и программы 445
Глава 6. Ой-ой-ой: объекты и классы 447
Глава 7. Работаем с данными профессионально 451
Глава 8. Данные должны куда-то попадать 458
Глава 9. Распутываем Всемирную паутину 462
Глава 10. Системы 463
Глава 11. Конкуренция и сети 465

Приложение Е. Вспомогательные материалы 473
Приоритет операторов 473
Строковые методы 474
Изменение регистра 474
Поиск 474
Изменение 474
Форматирование 475
Тип строки 475
Атрибуты модуля string 476

Мари, Карин, Тому и Рокси

Введение

Эта книга познакомит вас с языком программирования Python. Она предназначена для начинающих программистов, но даже если вы уже писали программы и хотите лишь добавить Python к списку доступных вам языков, издание «Простой Python. Современный стиль программирования» поможет в этом.

Книга представляет собой неторопливое введение, которое постепенно проведет вас от основ к множеству более углубленных тем. Я использовал смесь стилей учебника и поваренной книги, чтобы по очереди объяснить новые термины и идеи. Код, написанный на языке Python, включен даже в самые первые главы.

Несмотря на то что книга ориентирована на начинающих читателей, я включил в нее темы, которые могут показаться сложными, вроде баз данных NoSQL или библиотек передачи сообщений. Я выбрал их потому, что они помогут решить многие проблемы лучше, чем стандартные приемы. Вы загрузите и установите те внешние пакеты Python, которые пригодятся, когда «встроенные батарейки» не подойдут для вашего приложения. Пробовать что-то новое весело.

Я также включил в книгу несколько примеров того, чего делать **не нужно**, особенно если вы уже работали с другими языками программирования и пытаетесь адаптировать их стиль для Python. Не буду утверждать, что язык программирования Python идеален, — просто покажу вам, чего следует избегать.

Иногда я буду делать подобные врезки, когда что-то может быть непонятно или же существует более *питонский* способ сделать это.

Аудитория

Эта книга пригодится всем, кто заинтересован в изучении потенциально самого популярного языка программирования, независимо от того, изучали ли вы другие языки программирования ранее.

Краткое описание

В первых семи главах объясняются основы языка программирования Python, их нужно читать по порядку. В последующих главах показывается, как язык программирования Python используется в определенных областях, таких как Интернет,

базы данных, сети и т. д., их можно читать в любом порядке. В первых трех приложениях демонстрируется применение языка программирования Python в искусстве, бизнесе и науке. Далее вы узнаете, как установить Python 3, если у вас его нет. После этого идут ответы к упражнениям, расположенным в конце каждой главы, а затем несколько полезных списков.

Глава 1. Программы похожи на руководства по вязанию носков или жарке картошки. С помощью реальных программ, написанных на языке Python, демонстрируются синтаксис языка, его возможности и способы применения в реальном мире. При сравнении Python не проигрывает другим языкам, но он не идеален. Более старая версия Python (Python 2) уступает место более новой (Python 3). Если у вас установлен Python 2, установите на свой компьютер Python 3. Воспользуйтесь интерактивным интерпретатором, чтобы самостоятельно запустить примеры из этой книги.

Глава 2. В этой главе показываются простейшие типы данных, применяемые в языке программирования Python: булевы переменные, целые числа, числа с плавающей точкой и текстовые строки. Вы также изучите простейшую математику и текстовые операции.

Глава 3. Мы рассмотрим встроенные структуры данных более высокого уровня: списки, кортежи, словари и наборы. Вы будете пользоваться этими типами данных, как конструктором Lego, чтобы создавать более сложные структуры. Вы научитесь проходить по ним с помощью *итераторов* и *списковых включений*.

Глава 4. Здесь вы будете сплетать структуры данных из предыдущих глав со структурами кода, чтобы выполнять сравнение, выборку или повторение операций. Вы узнаете, как упаковывать код в функции и обрабатывать ошибки с помощью исключений.

Глава 5. В этой главе показывается, как перейти к более крупным структурам данных: модулям, пакетам и программам. Вы узнаете, где можно разместить код и данные, ввести и вывести данные, обработать различные варианты и исследуете стандартную библиотеку Python.

Глава 6. Если вы уже занимались объектно-ориентированным программированием на других языках, Python по сравнению с ними покажется вам более простым. В главе 6 объясняется, когда следует использовать объекты и классы, а когда лучше применить модули, списки или словари.

Глава 7. Научитесь профессионально управлять данными. Эта глава полностью посвящена текстовым и двоичным данным, особенностям использования символов стандарта Unicode, а также вопросам ввода-вывода.

Глава 8. Данные нужно где-то размещать. В этой главе вы начнете работать с простыми файлами, каталогами и файловыми системами. Далее узнаете, как управляться с простыми файловыми форматами вроде CSV, JSON и XML. Вы также научитесь сохранять и получать данные из реляционных баз данных и из современных хранилищ данных NoSQL.

Глава 9. Всемирной сети посвящена отдельная глава, где рассматриваются клиенты, серверы, извлечение данных, API и фреймворки. В главе 9 вы разработаете реальный сайт, используя параметры запроса и шаблоны.

Глава 10. Эта глава посвящена системному программированию. Здесь вы научитесь управлять программами, процессами и потоками, поработаете с датой и временем, автоматизируете выполнение некоторых задач системного администрирования.

Глава 11. Тема этой главы — сети, а именно: службы, протоколы и API. В качестве примеров рассматриваются как низкоуровневые сокеты, библиотеки обмена сообщениями и системы массового обслуживания, так и развертывание на облачных системах.

Глава 12. В этой главе содержатся советы для разработчиков, пишущих на языке программирования Python. Они касаются установки, использования IDE, тестирования, отладки, журналирования, контроля исходного кода и документации. Глава 12 также поможет вам найти и установить полезные пакеты сторонних разработчиков, упаковать свой код для повторного использования, а также узнать, где получить более подробную информацию.

Приложение А. В первом приложении рассматривается, что люди делают с помощью языка программирования Python в искусстве: графике, музыке, анимации и играх.

Приложение Б. Некоторые особенности языка программирования Python можно применить и для бизнеса: визуализацию данных (графики, графы и карты), безопасность и регулирование.

Приложение В. Язык программирования Python широко используется в научной деятельности: математике и статистике, физике, биологии и медицине. В приложении демонстрируются возможности инструментов NumPy, SciPy и Pandas.

Приложение Г. Если вы еще не установили Python 3 на свой компьютер, в этом приложении вы найдете информацию о том, как это сделать, независимо от того, какая операционная система у вас установлена: Windows, Mac OS/X, Linux или Unix.

Приложение Д. Здесь содержатся ответы на упражнения, приведенные в конце каждой главы. Не подглядывайте туда, пока не попробуете решить задачи самостоятельно.

Приложение Е. В этом приложении содержатся справочные данные.

Версии Python

Языки программирования со временем меняются — разработчики добавляют в них новые возможности, а также исправляют ошибки. Примеры этой книги написаны и протестированы для версии Python 3.3. Версия 3.4 вышла в то же время, когда и эта книга, и я расскажу вам о некоторых нововведениях. Если хотите узнать, что и когда было добавлено в язык программирования Python, посетите страницу https://docs.python.org/3/whatsnew/. Там представлена техническая информация. Она, возможно, покажется трудной для понимания, если вы только начинаете изучать Python, но может пригодиться в будущем, если вам нужно будет писать программы для компьютеров, на которых установлены другие версии Python.

Соглашения, принятые в этой книге

В этой книге приняты следующие шрифтовые соглашения.

Курсив

Им обозначаются новые термины и понятия.

Шрифт для названий

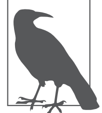
Применяется для отображения URL, адресов электронной почты, а также названий папок и выводимой на экран информации.

`Моноширинный шрифт`

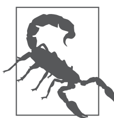
Используется в листингах программного кода, а также для имен и расширений файлов, названий путей, имен функций, команд, баз данных, переменных, операторов и ключевых слов.

`Курсивный моноширинный шрифт`

Указывает текст, который необходимо заменить пользовательскими значениями или значениями, определяемыми контекстом.



Этот рисунок указывает на совет, предложение или замечание.



Этот рисунок указывает на предупреждение.

Использование примеров кода

Примеры кода, приведенные в этой книге, — но не упражнения, которые являются заданиями для читателя, — доступны для загрузки по адресу https://github.com/madscheme/introducing-python. Эта книга написана, чтобы помочь вам при работе. В принципе, вы можете использовать код, содержащийся в ней, в ваших программах и документации. Можете не связываться с нами и не спрашивать разрешения, если собираетесь воспользоваться небольшим фрагментом кода. Например, если вы пишете программу и кое-где вставляете в нее код из книги, никакого особого разрешения не требуется. Однако если вы запишете на диск примеры из книги и начнете раздавать или продавать такие диски, то на это необходимо получить разрешение. Если вы цитируете это издание, отвечая на вопрос, или воспроизводите код из него в качестве примера, разрешение не нужно. Если вы включаете значительный фрагмент кода из данной книги в документацию по вашему продукту, необходимо разрешение.

Ссылки на источник приветствуются, но не обязательны. В такие ссылки обычно включаются название книги, имя ее автора, название издательства и номер ISBN. Например: *Introducing Python*, автор Билл Любанович (Bill Lubanovic). Copyright 2015 Bill Lubanovic, 978-1-449-35936-2.

При любых сомнениях относительно превышения разрешенного объема использования примеров кода, приведенных в данной книге, можете свободно обращаться к нам по адресу permissions@oreilly.com.

Как с нами связаться

Пожалуйста, направляйте комментарии и вопросы, связанные с этой книгой, ее издателю:

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

У нас есть веб-страница, посвященная этой книге, где мы размещаем опечатки, примеры и любую дополнительную информацию. Она располагается по адресу: http://bit.ly/introducing_python.

Чтобы оставить комментарий или задать технический вопрос об этой книге, отправляйте электронные письма по адресу bookquestions@oreilly.com.

Чтобы получить более подробную информацию о наших книгах, курсах, конференциях и новостях, посетите наш сайт http://www.oreilly.com.

Найдите нас на Facebook: http://facebook.com/oreilly.

Добавьте нас в свой Twitter: http://twitter.com/oreillymedia.

Смотрите нас на YouTube: http://www.youtube.com/oreillymedia.

Благодарности

Хочу объявить благодарность множеству людей, прочитавших и прокомментировавших мой черновик. В частности, я хотел бы упомянуть подробные обзоры Эли Бессерт (Eli Bessert), Генри Канивала (Henry Canival), Джереми Эллиота (Jeremy Elliott), Монте Миланука (Monte Milanuk), Лоика Пефферкорна (Loïc Pefferkorn) и Стивена Вейна (Steven Wayne).

Об авторе

Билл Любанович программировал в операционной системе Unix с 1977 года, разрабатывал GUI с 1981 года, базы данных с 1990 года, а веб-разработкой занимался с 1993 года.

В 1982 году, работая на стартапе Intran, он создал MetaForm — один из первых коммерчески успешных GUI (до Mac или Windows) для использования на одной из первых графических рабочих станций. В 1990 году он написал для компании Northwest Airlines визуальную систему управления доходами, которая дала миллионы долларов выручки. Кроме того, Любанович создал «витрину» компании в Интернете и написал для нее первый тест для анализа маркетинга в Сети. Позже, в 1994 году, он выступил сооснователем интернет-провайдера Tela, а в 1999 году участвовал в создании интернет-компании Mad Scheme.

Впоследствии Билл Любанович разрабатывал службы ядра и распределенные системы в составе команды, работающей на стартап с Манхэттена. В настоящее время автор этой книги занимается интеграцией сервисов OpenStack в суперкомпьютерной компании.

Билл счастливо живет в штате Миннесота со своей чудесной женой Мэри, сыном Томом и дочерью Карин, ухаживает за кошками Ингой и Люси и котом Честером.

1 Python: с чем его едят

Начнем с одной небольшой тайны и ее разгадки. Что, по-вашему, означают следующие две строки?

```
(Ряд 1): (RS) K18, ssk, k1, turn work.
(Ряд 2): (WS) Sl 1 pwise, p5, p2tog, p1, turn.
```

Выглядит как какая-то компьютерная программа. На самом деле это схема для вязания, а если точнее, фрагмент, который описывает, как связать пятку носка. Для меня эти строки имеют не больше смысла, чем кроссворд из газеты New York Times — для моего кота, но моя жена понимает их совершенно точно. Если вы вяжете, то тоже их поймете.

Рассмотрим еще один пример. Вы сразу поймете его предназначение, хотя и не сразу сможете определить результат:

```
½ столовой ложки масла или маргарина;
½ столовой ложки сливок;
2 ½ стакана муки;
1 чайная ложка соли;
1 чайная ложка сахара;
4 стакана картофельного пюре (охлажденного).
Перед тем как добавить муку, убедитесь, что все ингредиенты охлаждены.
Смешайте все ингредиенты.
Тщательно замесите.
Сделайте 20 шариков. Держите их охлажденными до следующего этапа.
Для каждого шарика разровняйте муку на тряпочке.
Раскатайте шарик при помощи рифленой скалки.
Жарьте на сковороде до подрумянивания.
Переверните и обжарьте другую сторону.
```

Даже если вы не готовите, вы сможете распознать *кулинарный рецепт*: список продуктов, за которым следуют указания по приготовлению. Но что получится в итоге? Это *лефсе*, норвежский деликатес, который напоминает тортилью. Полейте блюдо маслом, вареньем или чем-нибудь еще, сверните и наслаждайтесь.

Схема для вязания и рецепт имеют несколько похожих моментов:

- фиксированный словарь, состоящий из слов, аббревиатур и символов. Некоторые могут быть знакомы, другие же покрыты тайной;
- правила, описывающие, что и где можно говорить, — синтаксис;
- последовательность операций, которые должны быть выполнены по порядку;
- в некоторых случаях — повторение определенных операций *(цикл)*, например способ приготовления каждого кусочка лефсе;
- в некоторых случаях — ссылка на другую последовательность операций (говоря компьютерными терминами, *функция*). Например, когда вы прочтете приведенный выше рецепт, вам может понадобиться рецепт приготовления картофельного пюре;
- предполагаемое знание контекста. Рецепт подразумевает, что вы знаете, что такое вода и как ее кипятить. Схема для вязания подразумевает, что вы умеете держать спицы в руках;
- ожидаемый результат. В наших примерах результатом будет предмет для ног и предмет для желудка. Главное — не перепутать.

Все эти идеи вы можете встретить и в компьютерных программах. Я воспользовался этими «непрограммами», чтобы показать, что программы не так страшны, как может показаться. Нужно всего лишь выучить верные слова и правила.

Теперь оставим этих дублеров и рассмотрим настоящую программу. Что она делает?

```
for countdown in 5, 4, 3, 2, 1, "hey!":
    print(countdown)
```

Если вы считаете, что это программа, написанная на языке программирования Python, которая выводит на экран следующее:

```
5
4
3
2
1
hey!
```

то вы знаете, что язык программирования Python выучить проще, чем понять рецепт или схему для вязания. К тому же вы можете тренироваться писать на языке программирования Python, сидя за удобным и безопасным столом, избегая опасностей вроде горячей воды и спиц.

Программа, написанная на языке программирования Python, содержит несколько специальных слов и символов: `for`, `in`, `print`, запятые, точки с запятой, скобки

и т. д., — которые являются важной частью синтаксиса языка. Хорошая новость заключается в том, что язык программирования Python имеет более доступный и менее объемный синтаксис, чем большинство других языков программирования. Он кажется более понятным — почти как рецепт.

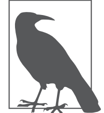
Вот еще одна небольшая программа, написанная на языке программирования Python, которая выбирает новостные клише из *списка* и выводит их на экран:

```
cliches = [
    "At the end of the day",
    "Having said that",
    "The fact of the matter is",
    "Be that as it may",
    "The bottom line is",
    "If you will",
    ]
print(cliches[3])
```

Эта программа выведет четвертое клише:

```
Be that as it may
```

Списки — вроде `cliches` — представляют собой последовательность значений, доступ к которым осуществляется с использованием *смещения* от начала списка. Смещение для первого элемента списка равно 0, а для четвертого — 3.



Люди считают с единицы, поэтому может показаться странным считать с нуля. При программировании удобнее оперировать смещениями, чем позициями.

Списки широко распространены в языке программирования Python. О том, как ими пользоваться, будет рассказано в главе 3.

Далее приведена еще одна программа, которая также выводит цитату, но в этот раз цитата выбирается в зависимости от того, кто ее произнес, а не с помощью позиции в списке:

```
quotes = {

    "Moe": "A wise guy, huh?",
    "Larry": "Ow!",
    "Curly": "Nyuk nyuk!",
    }
stooge = "Curly"
print(stooge, "says:", quotes[stooge])
```

Если вы запустите эту небольшую программу, она выведет следующее:

```
Curly says: Nyuk nyuk!
```

`quotes` — это *словарь*, коллекция уникальных *ключей* (в этом примере ключом является имя участника Stooge) и связанных с ними *значений* (в этом примере — значимая цитата участника Stooge). Используя словарь, вы можете сохранять элементы и выполнять их поиск по именам, что часто удобнее, чем работать со списком. Более подробно о словарях можно прочитать в главе 3.

В примере с клише для создания списка используются квадратные скобки ([и]), а в примере со Stooge для создания словаря — фигурные скобки ({ и }). Все это — варианты синтаксиса языка программирования Python, и в нескольких следующих главах вы увидите гораздо больше.

А теперь рассмотрим кое-что совершенно иное: в примере ниже показана программа, написанная на языке программирования Python, которая выполняет несколько более сложных задач. Не ждите, что сразу поймете, как она работает, — для этого и предназначена данная книга. Мы рассматриваем пример для того, чтобы увидеть и прочувствовать обычную нетривиальную программу, написанную на языке Python. Если вы знаете другие языки программирования, то можете сравнить их с Python прямо сейчас.

В примере ниже происходит подключение к сайту YouTube и получение информации о видеороликах, имеющих в данный момент самые высокие оценки. Если бы результатом была обычная веб-страница, заполненная текстом, отформатированным как HTML, было бы трудно получить всю необходимую информацию (я говорю об *извлечении данных* в разделе «Веб-сервисы и автоматизация» главы 9). Вместо этого пример получает данные, представленные в формате JSON, который предназначен для обработки компьютером. JSON, или JavaScript Object Notation, — это читабельный для человека текстовый формат, который описывает типы и значения, а также выстраивает значения в определенном порядке. Он немного похож на языки программирования и уже стал популярным способом обмена данными между разными языками программирования и системами. Вы можете прочитать о JSON больше в подразделе «JSON» раздела «Структурированные текстовые файлы» главы 8.

Программы, написанные на языке Python, могут преобразовывать текст формата JSON в *структуры данных* — их вы увидите в следующих двух главах, — как если бы вы написали программу, чтобы создавать их самостоятельно. В полученном от YouTube ответе данных очень много, поэтому в рамках этого примера я выведу названия лишь первых шести видеороликов. И вновь перед вами полноценная программа, которую вы можете запустить самостоятельно.

```
import json
from urllib.request import urlopen
url = "https://gdata.youtube.com/feeds/api/standardfeeds/top_rated?alt=json"
response = urlopen(url)
contents = response.read()
text = contents.decode('utf8')
data = json.loads(text)
for video in data['feed']['entry'][0:6]:
    print(video['title']['$t'])
```

Когда я запускал эту программу в последний раз, получил следующий результат:

```
Evolution of Dance — By Judson Laipply
Linkin Park — Numb
Potter Puppet Pals: The Mysterious Ticking Noise
"Chocolate Rain" Original Song by Tay Zonday
Charlie bit my finger — again !
The Mean Kitty Song
```

Эта небольшая программа, написанная на языке Python, делает многое с помощью всего лишь девяти строк. Если вы не знаете всех этих терминов, не волнуйтесь — вы познакомитесь с ними в следующих главах.

- Строка 1: импортируем весь код из *стандартной библиотеки*, которая называется json.
- Строка 2: импортируем только функцию `urlopen` из стандартной библиотеки urllib.
- Строка 3: присваиваем URL сайта YouTube переменной `url`.
- Строка 4: соединяемся с веб-сервером, расположенным по этому адресу, и запрашиваем определенный *веб-сервис*.
- Строка 5: получаем ответ и присваиваем его переменной `contents`.
- Строка 6: *дешифруем* содержимое переменной `contents` в текстовую строку формата JSON и присваиваем ее переменной `text`.
- Строка 7: преобразуем переменную `text` в `data` — структуру данных языка Python, предназначенную для работы с видео.
- Строка 8: получаем информацию для одного видеоролика единовременно в переменную `video`.
- Строка 9: используем двухуровневый словарь (`data['feed']['entry']`) и функцию `slice([0:6])`.
- Строка 10: используем функцию `print`, чтобы вывести на экран только название видеоролика.

Информация о видеоролике представляет собой различные структуры данных; все они демонстрируются в главе 3.

В предыдущем примере мы задействовали стандартные библиотечные модули (программы, включаемые в Python при установке), но в них нет ничего таинственного. Следующий фрагмент кода показывает переписанный пример, использующий внешний пакет ПО для Python, который называется `requests`:

```
import requests
url = "https://gdata.youtube.com/feeds/api/standardfeeds/top_rated?alt=json"
response = requests.get(url)
data = response.json()
for video in data['feed']['entry'][0:6]:
    print(video['title']['$t'])
```

Новая версия содержит всего шесть строк и, я полагаю, более читабельна для большинства людей. Я расскажу гораздо больше о `requests` и других авторских программах для Python в главе 5.

Python в реальном мире

Стоит ли тратить на изучение Python время и силы? Может быть, это игра в бирюльки? Язык программирования Python существует примерно с 1991 года (он появился раньше Java) и является одним из десяти самых популярных языков программирования. Людям платят деньги за то, что они пишут программы на Python, которыми мы пользуемся каждый день, — Google, YouTube, Dropbox, Netflix и Hulu. Я использовал Python для создания как поискового устройства для электронной почты, так и интернет-магазина. Python имеет репутацию высокопроизводительного языка программирования, что нравится динамично развивающимся организациям.

Вы можете найти множество приложений, написанных на Python, например:

- командную строку на мониторе или в окне терминала;
- пользовательские интерфейсы, включая сетевые;
- веб-приложения, как клиентские, так и серверные;
- бэкэнд-серверы, поддерживающие крупные популярные сайты;
- *облака* (серверы, управляемые сторонними организациями);
- приложения для мобильных устройств;
- приложения для встроенных устройств.

Программы, написанные на языке программирования Python, могут быть как одноразовыми *сценариями* — вы видели их ранее в этой главе, — так и сложными системами, содержащими миллионы строк. Мы рассмотрим применение языка программирования Python для создания сайтов, системного администрирования и манипулирования данными. Рассмотрим также использование Python в искусстве, науке и бизнесе.

Python против языка X

Насколько Python хорош по сравнению с другими языками программирования? Где и когда следует использовать тот или иной язык? В этом разделе я приведу примеры кода, написанные на других языках, чтобы вы могли понять, с чем конкурирует Python. Вы **не обязаны** понимать каждый из этих фрагментов, если не работали с этими языками. (Когда вы увидите последний фрагмент, написанный на Python, то почувствуете облегчение из-за того, что не работали с некоторыми

языками.) Если вам интересен только Python, вы ничего не потеряете, если не будете читать этот раздел.

Каждая программа должна вывести число и немного рассказать о языке, на котором она написана.

Если вы пользуетесь терминалом или терминальным окном, программа, которая читает то, что вы вводите, выполняет это и отображает результат, называется программой-*оболочкой*. Оболочка операционной системы Windows называется `cmd`, она выполняет *пакетные* файлы, имеющие расширение `.bat`. Для Linux и других операционных систем семейства Unix (включая Mac OS X) существует множество программ-оболочек, самая популярная из которых называется `bash` или `sh`. Оболочка обладает небольшими возможностями вроде выполнения простой логики и разворачивания символа-джокера наподобие * в полноценные имена файлов. Вы можете сохранять команды в файлы, которые называются *сценариями оболочки*, и выполнять их позже. Эти программы могли быть самыми первыми в вашей карьере программиста. Проблема заключается в том, что со сценариями оболочки трудно работать, если они содержат как минимум несколько сотен строк, а сами сценарии выполняются гораздо медленнее, чем программы, написанные на других языках. В следующем фрагменте кода демонстрируется небольшая программа-оболочка:

```
#!/bin/sh
language=0
echo "Language $language: I am the shell. So there."
```

Если вы сохраните этот файл под именем `meh.sh` и запустите его с помощью команды `sh meh.sh`, то на экране увидите следующее:

```
Language 0: I am the shell. So there.
```

Старые добрые C и C++ являются довольно низкоуровневыми языками программирования, которыми пользуются в том случае, когда важна скорость. Их труднее выучить, и вам придется отслеживать множество деталей, что может привести к падениям программы и проблемам, которые трудно диагностировать. Так выглядит небольшая программа на языке C:

```
#include <stdio.h>
int main(int argc, char *argv[]) {
    int language = 1;
    printf("Language %d: I am C! Behold me and tremble!\n", language);
    return 0;
}
```

C++ происходит из одного семейства с C, но имеет несколько отличительных особенностей:

```
#include <iostream>
using namespace std;
```

```
int main() {
    int language = 2;
    cout << "Language " << language << \
       ": I am C++!  Pay no attention to that C behind the curtain!" << \
       endl;
    return(0);
}
```

Java и C# являются преемниками языков C и C++, избавленными от некоторых проблем предшественников. Однако они немного избыточны и ограничительны. Следующий пример написан на Java:

```
public class Overlord {
    public static void main (String[] args) {
        int language = 3;
        System.out.format("Language %d: I am Java! Scarier than C!\n", language);
    }
}
```

Если вы никогда не писали программ ни на одном из этих языков, вам может быть интересно, **что все это такое**. Некоторые языки нагружены значительным синтаксическим багажом. Их иногда называют статическими языками, поскольку они требуют, чтобы вы указали компьютеру некоторые низкоуровневые детали. Позвольте мне объяснить.

Языки программирования имеют *переменные* — имена значений, которые вы хотите использовать в программе. Статические языки заставляют вас указывать *тип* каждой переменной, который определяет, сколько места переменная займет в памяти и что с ней можно сделать. Компьютер использует эту информацию, чтобы *скомпилировать* программу в очень низкоуровневый *машинный язык* (характерный для определенного аппаратного обеспечения, машины понимают его лучше, а люди — хуже). Дизайнеры языков программирования часто должны решать, кому их язык должен быть понятнее: людям или компьютерам. Объявление типов переменных помогает компьютеру найти некоторые ошибки и работать быстрее, но это требует предварительного продумывания и набора кода. Бóльшая часть кода примеров, написанного на языках C, C++ и Java, требует объявления типов переменных. Например, в каждом из примеров объявление типа `int` было необходимо для того, чтобы переменная `language` считалась целым числом. (Другие типы включают в себя числа с плавающей точкой, вроде `3.14159`, и символьные или текстовые данные, которые хранятся по-разному.)

Почему же они называются *статическими* языками? Потому что переменные в этих языках не могут изменять свой тип, они статичны. Целое число — это целое число, раз и навсегда.

Динамические языки — полная противоположность статических (они также называются *скриптовыми языками*). Эти языки программирования не заставляют вас определять тип переменной перед тем, как ее использовать. Если вы напишете

что-то вроде x = 5, динамический язык определит, что 5 — это целое число, поэтому переменная x имеет тип int. Эти языки позволяют вам достичь большего, написав меньшее количество строк кода. Вместо того чтобы компилироваться, они *интерпретируются* программой, которая называется — сюрприз! — *интерпретатором*. Динамические языки обычно медленнее, чем статические, но их скорость повышается, поскольку интерпретаторы становятся более оптимизированными. Долгое время динамические языки использовались для коротких программ *(сценариев)*, которые часто предназначались для того, чтобы подготовить данные для обработки более длинными программами, написанными на статических языках. Такие программы назывались *связующим кодом*. Несмотря на то что динамические языки больше годятся для этой цели, в наши дни они могут решать и самые трудные задачи по обработке данных.

Многоцелевым динамическим языком многие годы был Perl. Язык программирования Perl очень мощный и имеет множество библиотек. Однако его синтаксис может быть трудным для понимания, а сам язык теряет в популярности из-за появления языков программирования Python и Ruby. А вот извольте: острый код с привкусом Perl:

```
my $language = 4;
print "Language $language: I am Perl, the camel of languages.\n";
```

Язык программирования Ruby (http://www.ruby-lang.org/) появился немного позже. Он отчасти заимствует функционал у языка Perl, а свою популярность приобрел благодаря фреймворку для веб-разработки *Ruby on Rails*. Он используется примерно в тех же областях, что и Python, и, если выбирать между этими языками, вам придется руководствоваться в большей степени вкусом и доступными библиотеками. Следующий фрагмент кода написан на Ruby:

```
language = 5
puts "Language #{language}: I am Ruby, ready and aglow."
```

Язык программирования PHP (http://www.php.net/), который вы можете увидеть в следующем примере, очень популярен в области веб-разработки, поскольку позволяет довольно легко объединить HTML и код. Однако язык PHP имеет несколько подводных камней, и его довольно трудно применить за пределами веб-разработки:

```
<?PHP
$language = 6;
echo "Language $language: I am PHP. The web is <i>mine</i>, I say.\n";
?>
```

Следующий пример показывает ответ Python этим языкам программирования:

```
language = 7
print("Language %s: I am Python. What's for supper?" % language)
```

Почему же Python?

Python — многоцелевой высокоуровневый язык программирования. Его дизайн позволяет писать хорошо *читаемый* код, что гораздо важнее на деле, чем на словах. Каждая компьютерная программа пишется всего однажды, но впоследствии к ней обращаются множество раз. Удобочитаемость позволяет легко запомнить программу, а также легко *воспроизвести*. По сравнению с другими популярными языками программирования кривая обучения для языка Python более гладкая, что позволяет вам быстрее стать продуктивными. Однако есть и сложные моменты, которые вы можете исследовать по мере приобретения опыта.

Относительный лаконизм языка Python позволяет создать программу, которая будет гораздо короче своего аналога, написанного на статическом языке. Исследования показали, что программисты пишут примерно одинаковое количество строк кода каждый день независимо от языка, поэтому Python может значительно повысить вашу продуктивность. Язык программирования Python — самое несекретное оружие многих компаний, которым важна продуктивность работы сотрудников.

Python является самым популярным языком на курсах программирования для начинающих в лучших американских колледжах (http://bit.ly/popular-py). Он также используется для оценки навыков программирования более чем 2000 работодателей (http://bit.ly/langs-2014).

И конечно же, он абсолютно бесплатен. Вы можете написать с помощью Python все, что захотите, и пользоваться этой программой где угодно совершенно бесплатно. Никто не сможет прочитать вашу программу и сказать: «Какая милая программа! Будет жаль, если с ней что-то случится».

Python запускается практически везде и имеет «встроенные батарейки» — целую кучу полезного ПО в стандартных библиотеках.

Но, возможно, основная причина использования Python покажется вам неожиданной: людям обычно **нравится** этот язык. Им действительно нравится программировать на нем, а не относиться к нему как к еще одному инструменту. Некоторые разработчики говорят, что им не хватает какой-то особенности Python, когда они вынуждены программировать на другом языке. И это отличает Python от его «коллег».

Когда не стоит использовать Python

Python не всегда будет наилучшим выбором.

Он не предустановлен по умолчанию. В приложении Г показано, как установить Python, если он еще не установлен на вашем компьютере.

Python довольно быстрый для большинства приложений, но его скорости может оказаться недостаточно для наиболее требовательных из них. Если ваша программа проводит бо́льшую часть времени за вычислениями (в технических терминах такое называется *«ограничена быстродействием процессора»* (CPU-bound)),

то языки C, C++ или Java справятся с задачей гораздо лучше, чем Python. Но не всегда!

- Иногда более качественный *алгоритм* (пошаговое решение) для Python превосходит по скорости неэффективный алгоритм для C. Более высокая скорость разработки для Python дает вам больше времени для экспериментов над альтернативными решениями.
- Во многих приложениях программа «скрещивает пальцы» в ожидании ответа от сервера. Центральный процессор (компьютерный чип, который делает все расчеты) обычно не задействован, поэтому время выполнения статических и динамических программ будет примерно одинаковым.
- Стандартный интерпретатор Python написан на C и может быть улучшен с помощью дополнительного кода. Я рассмотрю этот вопрос в разделе «Оптимизируем ваш код» главы 12.
- Интерпретаторы для Python становятся быстрее. Java был ужасно медленным, когда только появился, и для его ускорения было потрачено много времени и денег. Языком программирования Python не владеет ни одна корпорация, поэтому он улучшается более плавно. В подразделе «PyPy» упомянутого раздела главы 12 я расскажу о проекте *PyPy* и его приложениях.
- Вы можете писать очень трудоемкое приложение, и, что бы вы ни делали, Python не будет соответствовать вашим потребностям. Тогда, как сказал Иен Холм в фильме «Чужой», примите мои соболезнования. Обычно альтернативой в таком случае являются языки программирования C, C++ и Java, однако решением может стать и более новый язык программирования — Go (http://golang.org/) (который, по ощущениям, похож на Python, но имеет более высокую производительность, вроде C).

Python 2 против Python 3

Самая большая проблема, с которой вы можете столкнуться сейчас, — это выбор одной из двух существующих версий Python. Python 2, кажется, существовал всегда, эта версия предустанавливается на компьютеры с операционными системами семейства Linux. Это был отличный язык, но ничто не идеально. В языках программирования, как и во многих иных областях, одни ошибки поверхностные, и исправить их легко, а другие — трудно. Решения этих трудных проблем **несовместимы**: новые программы, написанные с помощью исправленного языка, не будут работать на старых системах, а старые программы не будут работать на новых.

Создатель языка Python Гвидо ван Россум (Guido van Rossum) (https://www.python.org/~guido) и другие объединили решения трудных проблем и назвали их Python 3. Python 2 — это прошлое, а Python 3 — будущее. Последняя версия Python 2 имеет номер 2.7, она еще долго будет поддерживаться, но на ней род заканчивается; Python 2.8 никогда не выйдет. Новая разработка будет вестись на Python 3.

В этой книге описывается Python 3. Если вы раньше использовали Python 2, то практически не заметите разницы. Самое очевидное изменение — это способ вызова функции `print`. Самое главное изменение — это обработка символов *Unicode*, она рассматривается в главах 2 и 7. Преобразование популярного ПО, написанного на Python, выполняется постепенно. Но сейчас кажется, что мы наконец достигли переломного момента.

Установка Python

Чтобы не занимать много места, я вынес детали установки Python 3 в приложение Г. Если у вас еще не установлен Python 3 или вы не знаете этого точно, обратитесь к приложению и посмотрите, что вам нужно сделать со своим компьютером.

Запуск Python

После того как вы установите рабочую копию Python 3, можете использовать ее, чтобы запускать как программы, приведенные в этой книге, так и собственный код. Как же запустить программу, написанную на языке Python? Существует два основных способа.

- *Интерактивный интерпретатор*, который поставляется вместе с Python, дает возможность экспериментировать с небольшими программами. Вы вводите команды строка за строкой и мгновенно видите результат. Благодаря тесному связыванию между печатанием и просмотром можете проводить эксперименты быстрее. Я буду использовать интерактивный интерпретатор, чтобы продемонстрировать возможности языка, а вы можете вводить те же команды в собственной среде Python.
- Для всего прочего сохраняйте программы в виде текстовых файлов с расширением `.py`, а затем запускайте их, введя `python` и имена этих файлов.

Попробуем воспользоваться обоими методами.

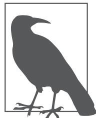
Интерактивный интерпретатор

Для большинства примеров кода в этой книге используется интерактивный интерпретатор. Когда вы вводите команды, которые видите в примерах, и получаете те же результаты, вы знаете, что идете по правильному пути.

Интерпретатор запускается путем ввода имени основной программы Python для вашего компьютера: `python`, `python3` или чего-то похожего. В дальнейшем мы будем предполагать, что она называется `python`. Если ваша программа называется по-другому, то для ее запуска вам следует ввести именно это имя.

Интерактивный интерпретатор работает практически так же, как и интерпретатор для файлов, но с одним исключением: когда вы вводите обычное значение,

интерактивный интерпретатор автоматически выведет его на экран. Например, если вы запустите Python и введете в интерпретатор число 61, оно будет продублировано в терминале.



В следующем примере символ $ — это обычное приглашение ввести команду вроде python в окно терминала. Мы будем использовать ее для примеров кода в этой книге, однако ваше приглашение может отличаться.

```
$ python
Python 3.3.0 (v3.3.0:bd8afb90ebf2, Sep 29 2012, 01:25:11)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> 61
61
>>>
```

Автоматическое выведение значения — это экономящая время особенность интерактивного интерпретатора, а не часть языка Python.

Кстати, функция `print()` также работает внутри интерпретатора, на случай если вам понадобится вывести что-то на экран:

```
>>> print(61)
61
>>>
```

Если вы попробовали запустить эти примеры с помощью интерактивного интерпретатора и увидели те же результаты, то у вас появился опыт (пусть и небольшой) запуска кода на Python. В следующих нескольких главах вы перейдете от строковых команд к более длинным программам.

Файлы Python

Если вы запишете в файл число `61` и запустите этот файл с помощью Python, он выполнится, но на экране ничего не появится. В обычных неинтерактивных программах для Python вам нужно вызвать функцию `print`, чтобы вывести что-то на экран, как показано в следующем фрагменте кода:

```
print(61)
```

Создадим файл программы Python и запустим его.

1. Откройте текстовый редактор.
2. Введите в него строку `print(61)`, как показано ранее.
3. Сохраните этот файл с именем `61.py`. Убедитесь, что вы сохранили его как простой текст, а не в формате вроде RTF или DOC. Вы не обязаны использовать расширение `.py` для файлов программ Python, но оно поможет вам запомнить предназначение файла.

4. Если вы пользуетесь графическим пользовательским интерфейсом — это касается практически каждого, — откройте окно терминала[1].
5. Запустите программу, введя следующую строку:

```
$ python 61.py
```

Вы должны увидеть такую строку:

```
61
```

Сработало? Если да, то примите мои поздравления по поводу того, что вы запустили свою первую автономную программу для Python.

Что дальше?

Вы будете вводить команды в работающую систему Python, они должны соответствовать синтаксису языка. Вместо того чтобы вывалить все синтаксические правила сразу, рассмотрим их в нескольких следующих главах.

Самый простой способ разработки программы на Python — применение простого текстового редактора и окна терминала. В рамках этой книги я использую именно такие редакторы, иногда показывая интерактивные сессии работы с терминалом, а иногда — фрагменты файлов. Вам следует знать, что существует множество *интегрированных сред разработки* (Integrated Development Environment, IDE) для Python. Они могут предоставлять вам графические пользовательские интерфейсы, помогающие в редактировании текста, и экраны помощи. Более подробно вы прочитаете о них в главе 12.

Момент просветления

Каждый язык программирования имеет свой стиль. Во введении я упомянул, что существует *характерный для Python* способ выразить себя. В Python встроен небольшой текст, который выражает его философию (насколько я знаю, Python — это единственный язык программирования, содержащий подобное «пасхальное яйцо»). Когда вам захочется ощутить момент просветления, просто введите `import this` в интерактивный интерпретатор, а затем нажмите клавишу Enter:

```
>>> import this
Красивое лучше, чем уродливое.
Явное лучше, чем неявное.
Простое лучше, чем сложное.
Сложное лучше, чем запутанное.
Одноуровневое лучше, чем вложенное.
Разреженное лучше, чем плотное.
```

[1] Если вы не знаете, что это значит, откройте приложение Г, чтобы получить детальную информацию для различных операционных систем.

```
Читаемость имеет значение.
Особые случаи не настолько особые, чтобы нарушать правила.
При этом практичность важнее безупречности.
Ошибки никогда не должны замалчиваться.
Если не замалчиваются явно.
Встретив двусмысленность, отбрось искушение угадать.
Должен существовать один — и желательно только один — очевидный способ сделать это.
Хотя он поначалу может быть и не очевиден, если вы не голландец.
Сейчас лучше, чем никогда.
Хотя никогда зачастую лучше, чем прямо сейчас.
Если реализацию сложно объяснить — идея плоха.
Если реализацию легко объяснить — идея, возможно, хороша.
Пространства имен — отличная штука! Будем делать их побольше!
```

На протяжении книги я буду приводить примеры, иллюстрирующие эти утверждения.

Упражнения

Эта глава является введением в язык программирования Python. Здесь было показано, что он делает, как выглядит и где его можно применить. В конце каждой главы я буду предлагать выполнить небольшие проекты, которые помогут вам запомнить то, что вы только что прочитали, и подготовят к следующим урокам.

1. Если вы еще не установили Python 3, сделайте это сейчас. Прочтите приложение Г, чтобы узнать детали.
2. Запустите интерактивный интерпретатор Python 3. И вновь детали вы найдете в приложении Г. Интерпретатор должен вывести несколько строк о себе, а затем строку, начинающуюся с символов >>>. Перед вами приглашение для ввода команд Python.
3. Немного поэкспериментируйте с интерпретатором. Используйте его как калькулятор и наберите текст `8 * 9`. Нажмите клавишу **Enter**, чтобы увидеть результат. Python должен вывести `72`.
4. Теперь введите число `47` и нажмите клавишу **Enter**. Появилось ли число `47` в следующей строке?
5. Теперь введите `print(47)` и нажмите клавишу **Enter**. Появилось ли снова число `47` в следующей строке?

2 Ингредиенты Python: числа, строки и переменные

В этой главе мы рассмотрим простейшие встроенные в Python типы данных:

- *булевы значения* (которые имеют значение `True` или `False`);
- *целые числа* (вроде 42 и 100 000 000);
- *числа с плавающей точкой* (числа с десятичной запятой, вроде 3,14159 или экспоненты, вроде 1,0e8, что означает «один умножить на десять в восьмой степени», или 100 000 000,0);
- *строки* (последовательности текстовых символов).

Можно сказать, что они являются атомами. В этой главе мы будем использовать их обособленно. В главе 3 будет показано, как объединить их в молекулы.

Каждый тип имеет специфические правила использования, они по-разному обрабатываются компьютером. Мы также познакомимся с *переменными* (имена, которые ссылаются на данные; чуть подробнее мы поговорим о них совсем скоро).

Примеры кода, приведенные в этой главе, корректны с точки зрения Python, но они являются лишь фрагментами кода. Мы будем использовать интерактивный интерпретатор Python, вводя в него эти фрагменты и немедленно получая результат. Попробуйте запустить их самостоятельно. Вы распознаете эти примеры по приглашению >>>. В главе 4 мы начнем писать программы, которые могут работать самостоятельно.

Переменные, имена и объекты

В Python все — булевы значения, целые числа, числа с плавающей точкой, строки и даже крупные структуры данных, функции и программы — реализовано как объект. Это позволяет языку быть стабильным (и дает полезные особенности), чего не хватает некоторым другим языкам.

Объект похож на прозрачный пластиковый ящик, который содержит фрагмент данных (рис. 2.1). Объект имеет тип вроде булевых значений или целых чисел, который определяет, что можно сделать с этими данными. В реальном мире ящик с надписью «Керамика» может сообщить некоторую информацию (скорее всего, он тяжелый и лучше не ронять его на пол). Точно так же и в Python — если объект имеет тип `int`, вы знаете, что сможете сложить его с другим объектом типа `int`.

Рис. 2.1. Объект похож на коробку

Тип также определяет, можно ли изменить значение, которое хранится в ящике (изменяемое значение), или оно константно (неизменяемое значение). Неизменяемый объект можно сравнить с закрытым ящиком с окошком: вы можете увидеть значение, но не можете изменить его. В рамках той же аналогии изменяемый объект похож на открытую коробку: вы не только можете увидеть хранящееся там значение, но и изменить его, однако не можете изменить его тип.

Python является строго типизированным языком, а это означает, что тип объекта не изменится, даже если можно поменять его значение (рис. 2.2).

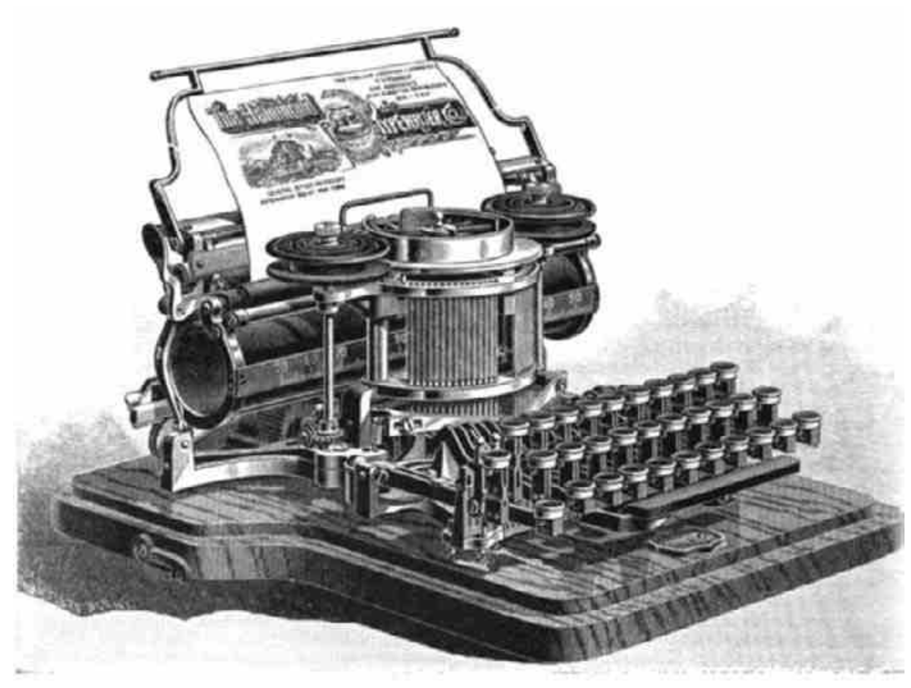
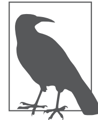


Рис. 2.2. Строгая типизация не означает, что нужно нажимать клавиши со строгим выражением лица

Языки программирования также позволяют вам определять переменные. Переменные являются именами, которые ссылаются на значения в памяти компьютера. Вы можете определить их для использования в своей программе. В Python символ = применяется для присваивания значения переменной.



В школе нас всех учили, что символ = означает «равно». Почему же во многих языках программирования, включая Python, этот символ используется для обозначения присваивания? Одна из причин — на стандартной клавиатуре отсутствуют логические альтернативы вроде стрелки влево, а символ = не слишком сбивает с толку. Кроме того, в компьютерных программах присваивание используется чаще проверки на равенство.

В следующей программе целое число 7 присваивается переменной с именем a, затем на экран выводится значение, связанное в текущий момент с этой переменной:

```
>>> a = 7
>>> print(a)
7
```

Сейчас пришло время сделать очень важное заявление о переменных в Python: переменные — это просто имена. Присваивание **не копирует** значение, оно **прикрепляет имя** к объекту, который содержит данные. Имя — это *ссылка* на какой-то объект, а не сам объект. Имя можно рассматривать как стикер (рис. 2.3).

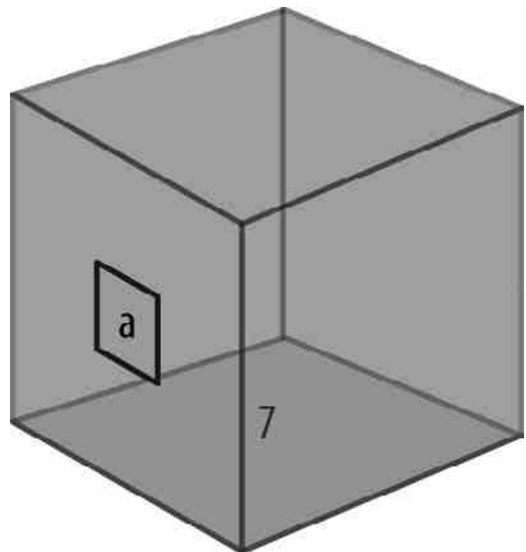


Рис. 2.3. Имена прикрепляются к объектам

Попробуйте сделать следующее с помощью интерактивного интерпретатора.

1. Как и раньше, присвойте значение 7 имени a. Это создаст объект-«ящик», содержащий целочисленное значение 7.

2. Выведите на экран значение a.
3. Присвойте a переменной b, заставив b прикрепиться к объекту-«ящику», содержащему значение 7.
4. Выведите значение b.

```
>>> a = 7
>>> print(a)
7
>>> b = a
>>> print(b)
7
```

В Python, если вы хотите узнать тип какого-то объекта (переменной или значения), вам следует использовать конструкцию type(объект). Попробуем сделать это для разных значений (58, 99.9, abc) и переменных (a, b):

```
>>> type(a)
<class 'int'>
>>> type(b)
<class 'int'>
>>> type(58)
<class 'int'>
>>> type(99.9)
<class 'float'>
>>> type('abc')
<class 'str'>
```

Класс — это определение объекта; классы детально рассматриваются в главе 6. В Python значения терминов «класс» и «тип» примерно одинаковы.

Имена переменных могут содержать только следующие символы:

- буквы в нижнем регистре (от «a» до «z»);
- буквы в верхнем регистре (от «A» до «Z»);
- цифры (от 0 до 9);
- нижнее подчеркивание (_).

Имена не могут начинаться с цифры. Python также особо обрабатывает имена, которые начинаются с нижнего подчеркивания (об этом вы можете прочесть в главе 4). Корректными являются следующие имена:

- a;
- a1;
- a_b_c___95;
- _abc;
- _1a.

Следующие имена, однако, некорректны:

- 1;
- 1a;
- 1_.

Наконец, не следует использовать следующие слова для имен переменных, поскольку они являются *зарезервированными словами* Python:

false	class	finally	is	return
none	continue	for	lambda	try
true	def	from	nonlocal	while
and	del	global	not	with
as	elif	if	or	yield
assert	else	import	pass	
break	except	in	raise	

Эти слова и некоторые знаки препинания используются в синтаксисе Python. Вы познакомитесь с ними всеми по мере чтения этой книги.

Числа

Python имеет встроенную поддержку целых чисел (наподобие 5 и 1 000 000 000) и чисел с плавающей точкой (вроде 3,1416, 14,99 и 1,87e4). Вы можете вычислять комбинации чисел с помощью простых математических операторов, приведенных в таблице.

Оператор	**Описание**	**Пример**	**Результат**
+	Сложение	5 + 8	13
–	Вычитание	90 – 10	80
*	Умножение	4 * 7	28
/	Деление с плавающей точкой	7/2	3,5
//	Целочисленное (Truncating) деление	7//2	3
%	Modulus (вычисление остатка)	7%3	1
**	Возведение в степень	3^4	81

На нескольких следующих страницах я покажу вам простые примеры того, как Python можно использовать в качестве очень сложного калькулятора.

Целые числа

Любая последовательность цифр в Python считается *целым числом*:

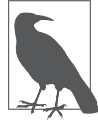
```
>>> 5
5
```

Можно использовать и простой ноль (0):

```
>>> 0
0
```

Но не ставьте его перед другими цифрами:

```
>>> 05
  File "<stdin>", line 1
    05
     ^
SyntaxError: invalid token
```



Только что вы увидели первое исключение в Python — программную ошибку. В нашем случае это предупреждение о том, что значение 05 — это invalid token (некорректный символ). Я объясню, что это значит, в подразделе «Системы счисления» далее. В этой книге вы увидите еще много примеров исключений, поскольку они являются основным механизмом обработки ошибок в Python.

Последовательность цифр указывает на целое число. Если вы поместите знак + перед цифрами, число останется прежним:

```
>>> 123
123
>>> +123
123
```

Чтобы указать на отрицательное число, вставьте перед цифрами знак -:

```
>>> -123
-123
```

С помощью Python вы можете выполнять обычные арифметические действия, как и с обычным калькулятором, используя операторы, показанные в предыдущей таблице. Сложение и вычитание будут работать, полностью соответствуя вашим ожиданиям:

```
>>> 5 + 9
14
>>> 100 – 7
93
>>> 4 – 10
-6
```

Вы можете работать с любым количеством чисел и операторов:

```
>>> 5 + 9 + 3
17
>>> 4 + 3 – 2 – 1 + 6
10
```

Замечание по стилю: не обязательно вставлять пробел между каждым числом и оператором:

```
>>> 5+9   +      3
17
```

Такой формат выглядит лучше, и его проще прочесть.

Умножение тоже довольно привычно:

```
>>> 6 * 7
42
>>> 7 * 6
42
>>> 6 * 7 * 2 * 3
252
```

Операция деления чуть более интересна, поскольку существует два ее вида:

- с помощью оператора / выполняется *деление с плавающей точкой* (десятичное деление);
- с помощью оператора // выполняется *целочисленное деление* (деление с остатком).

Даже если вы делите целое число на целое число, оператор / даст результат с плавающей точкой:

```
>>> 9 / 5
1.8
```

Целочисленное деление даст вам целочисленный ответ, отбрасывая остаток:

```
>>> 9 // 5
1
```

Деление на ноль с помощью любого оператора сгенерирует исключение:

```
>>> 5 / 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>> 7 // 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: integer division or modulo by z
```

Во всех предыдущих примерах используются непосредственно целочисленные значения. Вы можете смешивать целочисленные значения и переменные, которым было присвоено целочисленное значение:

```
>>> a = 95
>>> a
95
>>> a - 3
92
```

Ранее, когда мы выполнили операцию a - 3, мы не присвоили результат переменной a, поэтому ее значение не изменилось:

```
>>> a
95
```

Если вы хотите изменить значение переменной a, придется сделать следующее:

```
>>> a = a — 3
>>> a
92
```

Это обычно сбивает с толку начинающих программистов, потому что благодаря изучению математики в школе мы видим знак = и думаем, что он указывает на равенство. В Python выражение, стоящее справа от знака =, вычисляется первым **и только затем** присваивается переменной с левой стороны.

Проще думать об этом так.

1. Вычитаем 3 из a.
2. Присваиваем результат этого вычитания временной переменной.
3. Присваиваем значение временной переменной a:

```
>>> a = 95
>>> temp = a — 3
>>> a = temp
```

Поэтому, когда вы говорите:

```
>>> a = a — 3
```

Python рассчитывает результат операции вычитания с правой стороны от знака =, запоминает результат, а затем присваивает его переменной a, которая находится с левой стороны. Это гораздо быстрее и приятнее глазу, чем использование временной переменной.

Вы можете совместить арифметические операторы с присваиванием, размещая оператор перед знаком =. В этом примере выражение a -= 3 аналогично выражению a = a - 3:

```
>>> a = 95
>>> a -= 3
>>> a
92
```

Это выражение аналогично выражению a = a + 8:

```
>>> a += 8
>>> a
100
```

А это — выражению a = a * 2:

```
>>> a *= 2
>>> a
200
```

Здесь представлен пример деления с плавающей точкой, a = a / 3:

```
>>> a /= 3
>>> a
66.66666666666667
```

Присвоим значение 13 переменной a, а затем попробуем использовать сокращенный вариант a = a // 4 (целочисленное деление):

```
>>> a = 13
>>> a //= 4
>>> a
3
```

Символ % имеет несколько разных применений в Python. Когда он находится между двух чисел, с его помощью вычисляется остаток от деления первого числа на второе:

```
>>> 9 % 5
4
```

Вот так можно получить частное и остаток одновременно:

```
>>> divmod(9,5)
(1, 4)
```

В противном случае вам пришлось бы считать их по отдельности:

```
>>> 9 // 5
1
>>> 9 % 5
4
```

Только что вы увидели кое-что новое: *функцию* с именем divmod, в которую передаются целые числа 9 и 5, возвращающую двухэлементный результат, называемый *кортежем*. С кортежами вы познакомитесь в главе 3, а с функциями — в главе 4.

Приоритет операций

Рассмотрим, что получится, если ввести следующее:

```
>>> 2 + 3 * 4
```

Если выполнить сложение первым, 2 + 3 равно 5, а 5 * 4 равно 20. Но если выполнить первым умножение, 3 * 4 равно 12, а 2 + 12 равно 14. В Python, как и в большинстве

других языков, умножение имеет больший *приоритет*, нежели сложение, поэтому вы увидите ответ, совпадающий со второй версией:

```
>>> 2 + 3 * 4
14
```

Как узнать приоритет той или иной операции? В приложении Е приводится огромная таблица, в которой перечислены все приоритеты, но я обнаружил, что на практике никогда не смотрю в эти правила. Гораздо проще добавить пару скобок, чтобы сгруппировать код и вычисления так, как нужно:

```
>>> 2 + (3 * 4)
14
```

Это поможет любому человеку, читающему код, точно определить его предназначение.

Системы счисления

Предполагается, что целые числа указываются в десятичной *системе счисления*, если только вы не укажете какую-либо другую. Вам может никогда не понадобиться использовать другие системы счисления, но иногда они будут встречаться в коде.

Как правило, у нас десять пальцев на руках и ногах (у одного из моих котов их немного больше, но он редко использует их для счета), поэтому мы считаем: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9. После этого у нас заканчиваются цифры и мы переносим единицу на место десятки и ноль — на место единицы: 10 означает *«одна десятка и ноль единиц»*. У нас нет одной цифры, которая представляла бы собой 10. Далее мы считаем: 11, 12... 19, переносим единицу, чтобы сделать 20 *(две десятки и ноль единиц)*, и т. д.

Система счисления указывает, сколько цифр вы можете использовать до того, как перенести единицу. В двоичной (бинарной) системе счисления единственными цифрами являются 0 и 1. Двоичные 0 и 1 точно такие же, как и десятичные. Однако, если в этой системе сложить 1 и 1, вы получите 10 (одна десятичная двойка плюс ноль десятичных единиц).

В Python вы можете выразить числа в трех системах счисления помимо десятичной:

- `0b` или `0B` для *двоичной системы* (основание 2);
- `0o` или `0O` для *восьмеричной системы* (основание 8);
- `0x` или `0X` для *шестнадцатеричной системы* (основание 16).

Интерпретатор выведет эти числа как десятичные. Попробуем воспользоваться каждой из систем счисления. Первой выберем старое доброе десятичное число 10, которое означает *«одна десятка и ноль единиц»*:

```
>>> 10
10
```

Теперь возьмем двоичную (основание 2), что означает *«одна (десятичная) двойка и ноль единиц»*:

```
>>> 0b10
2
```

Восьмеричная (основание 8) означает *«одна (десятичная) восьмерка и ноль единиц»*:

```
>>> 0o10
8
```

Шестнадцатеричная (основание 16) означает *«одна (десятичное) 16 и ноль единиц»*:

```
>>> 0x10
16
```

Если вам интересно, какие «цифры» использует шестнадцатеричная система счисления, взгляните на них: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e и f. 0xa равно десятичной 10, а 0xf — десятичному 15. Добавьте 1 к 0xf, и вы получите 0x10 (десятичное 16).

Зачем использовать другие системы счисления, отличные от десятичной? Это полезно для *битовых операций*, которые описаны в главе 7 наряду с детальной информацией о преобразовании чисел из одной системы счисления в другую.

Преобразования типов

Для того чтобы изменить другие типы данных на целочисленный тип, следует использовать функцию `int()`. Она сохраняет целую часть числа и отбрасывает любой остаток.

Простейший тип данных в Python — *булевы переменные*, значениями этого типа могут быть только `True` или `False`. При преобразовании в целые числа они представляют собой значения 1 и 0:

```
>>> int(True)
1
>>> int(False)
0
```

Преобразование числа с плавающей точкой в целое число просто отсекает все, что находится после десятичной запятой[1]:

```
>>> int(98.6)
98
>>> int(1.0e4)
10000
```

[1] В программах ставится точка.

Наконец, рассмотрим пример преобразования текстовой строки (со строками вы познакомитесь через несколько страниц), которая содержит только цифры и, возможно, знаки + и -:

```
>>> int('99')
99
>>> int('-23')
-23
>>> int('+12')
12
```

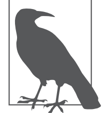
Преобразование целого числа в целое число ничего не меняет и совсем не вредит:

```
>>> int(12345)
12345
```

Если вы попробуете преобразовать что-то непохожее на число, сгенерируется *исключение*:

```
>>> int('99 bottles of beer on the wall')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: '99 bottles of beer on the wall'
>>> int('')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: ''
```

Текстовая строка в предыдущем примере начинается с корректных символов-цифр (99), но продолжается теми символами, которые функция int() обработать не может.



Мы рассмотрим исключения в главе 4. Пока нужно только помнить, что с помощью исключений Python извещает вас о том, что произошла ошибка, вместо того чтобы прервать выполнение программы, как поступают некоторые другие языки. Вместо того чтобы показывать вам лишь правильные примеры, я продемонстрирую множество вариантов исключений, чтобы вы знали, как поступает Python, когда что-то идет не так.

Функция int() будет создавать целые числа из чисел с плавающей точкой или строк, состоящих из цифр, но она не будет обрабатывать строки, содержащие десятичные точки или экспоненты:

```
>>> int('98.6')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: '98.6'
>>> int('1.0e4')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: '1.0e4'
```

Если вы смешаете численные значения, Python будет пытаться автоматически преобразовать их:

```
>>> 4 + 7.0
11.0
```

Булево значение `False` рассматривается как `0` или `0.0`, когда оно смешивается с целыми числами или числами с плавающей точкой, а `True` — как `1` или `1.0`:

```
>>> True + 2
3
>>> False + 5.0
5.0
```

Насколько объемен тип int?

В Python 2 размер переменной типа `int` был ограничен 32 битами. Этого было достаточно для того, чтобы сохранить любое целое число в диапазоне от –2 147 483 648 до 2 147 483 647.

Для переменных типа `long` выделялось еще больше места — 64 бита, что позволяло хранить значения в диапазоне от –9 223 372 036 854 775 808 до 9 223 372 036 854 775 807. В Python 3 тип `long` отсутствует, а переменная типа `int` может быть любого размера — даже больше 64 бит. Поэтому вы можете выполнить арифметические действия наподобие следующих (`10**100` называется *гугол* (googol), компания Google называлась так до того, как решила сменить свое название на более простое):

```
>>>
>>> googol = 10**100
>>> googol
100000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000
>>> googol * googol
100000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000
```

Во многих языках программирования этот пример вызовет так называемое *переполнение целочисленного значения*, когда числу нужно больше места, чем компьютер может предоставить, что приведет к разным нежелательным эффектам. Python обрабатывает огромные целые числа без всяких проблем. Одно очко для Python.

Числа с плавающей точкой

Значениями целочисленного типа являются целые числа, но *числа с плавающей точкой* (в Python они называются float) имеют десятичную запятую. Числа с плавающей точкой обрабатываются так же, как и целые: вы можете использовать операторы `+`, `-`, `*`, `/`, `//`, `**`, `%` и функцию `divmod()`.

Для того чтобы преобразовать другие типы в тип `float`, следует использовать функцию `float()`. Как и ранее, булевы значения обрабатываются как небольшие числа:

```
>>> float(True)
1.0
>>> float(False)
0.0
```

Преобразование значения типа `int` в тип `float` лишь создаст счастливого обладателя десятичной запятой:

```
>>> float(98)
98.0
>>> float('99')
99.0
```

Вы также можете преобразовывать строки, содержащие символы, которые являются корректным числом с плавающей точкой (цифры, знаки, десятичная запятая или `e`, за которой следует экспонента):

```
>>> float('98.6')
98.6
>>> float('-1.5')
-1.5
>>> float('1.0e4')
10000.0
```

Математические функции

Python имеет привычный набор математических функций вроде квадратного корня, косинуса и т. д. Мы рассмотрим их в приложении В, где также обсудим применение Python в науке.

Строки

Непрограммисты думают, что программисты хорошо разбираются в математике, потому что работают с числами. На самом деле большинство программистов работают с текстовыми *строками* гораздо чаще, чем с числами. Логическое (и креативное!) мышление для них зачастую гораздо важнее математических навыков.

Благодаря поддержке стандарта Unicode Python 3 может содержать символы любого языка мира, а также многие другие символы. Необходимость работы с этим стандартом была одной из причин изменения Python 2. Это хорошая причина использовать версию 3. Я буду применять стандарт Unicode лишь иногда, поскольку

это может показаться сложным. В следующих примерах я буду использовать строки формата ASCII.

Строки являются первым примером *последовательностей* в Python. В частности, они представляют собой последовательности символов.

В отличие от других языков, в Python строки являются *неизменяемыми*. Вы не можете изменить саму строку, но можете скопировать части строк в другую строку, чтобы получить тот же эффект.

Скоро вы узнаете, как это делается.

Создаем строки с помощью кавычек

Строка в Python создается заключением символов в одинарные или двойные кавычки, как показано в следующем примере:

```
>>> 'Snap'
'Snap'
>>> "Crackle"
'Crackle'
```

Интерактивный интерпретатор выводит на экран строки в одинарных кавычках, но все они обрабатываются одинаково.

Зачем иметь два вида кавычек? Основная идея заключается в том, что вы можете создавать строки, содержащие кавычки. Внутри одинарных кавычек можно расположить двойные и наоборот:

```
>>> "'Nay,' said the naysayer."
"'Nay,' said the naysayer."
>>> 'The rare double quote in captivity: ".'
'The rare double quote in captivity: ".'
>>> 'A "two by four" is actually 1 1/2" × 3 1/2".'
'A "two by four is" actually 1 1/2" × 3 1/2".'
>>> "'There's the man that shot my paw!' cried the limping hound."
"'There's the man that shot my paw!' cried the limping hound."
```

Можно также использовать три одинарные (`'''`) или три двойные кавычки (`"""`):

```
>>> '''Boom!'''
'Boom'
>>> """Eek!"""
'Eek!'
```

Тройные кавычки не очень полезны для таких коротких строк. Они обычно используются для того, чтобы создать *многострочные строки*, наподобие следующего классического стихотворения Эдварда Леара (Edward Lear):

```
>>> poem =  '''There was a Young Lady of Norway,
... Who casually sat in a doorway;
```

```
... When the door squeezed her flat,
... She exclaimed, "What of that?"
... This courageous Young Lady of Norway.'''
>>>
```

(Это стихотворение было введено в интерактивный интерпретатор, который поприветствовал нас символами >>> в первой строке и выводил символы ... до тех пор, пока мы не ввели последние тройные кавычки и не перешли к следующей строке.)

Если бы вы попробовали создать стихотворение с помощью одинарных кавычек, Python начал бы волноваться, когда бы вы перешли к следующей строке:

```
>>> poem = 'There was a young lady of Norway,
  File "<stdin>", line 1
    poem = 'There was a young lady of Norway,
                                            ^

SyntaxError: EOL while scanning string literal
>>>
```

Если внутри тройных кавычек располагается несколько строк, символы конца строки будут сохранены внутри нее. Если перед строкой или после нее находятся пробелы, они также будут сохранены:

```
>>> poem2 = '''I do not like thee, Doctor Fell.
...     The reason why, I cannot tell.
...     But this I know, and know full well:
...     I do not like thee, Doctor Fell.
... '''
>>> print(poem2)
I do not like thee, Doctor Fell.
    The reason why, I cannot tell.
    But this I know, and know full well:
    I do not like thee, Doctor Fell.
>>>
```

Кстати, существует разница между выводом на экран с помощью функции `print()` и автоматическим выводом на экран с помощью интерактивного интерпретатора:

```
>>> poem2
'I do not like thee, Doctor Fell.\n    The reason why, I cannot tell.\n    But
this I know, and know full well:\n    I do not like thee, Doctor Fell.\n'
```

Функция `print()` извлекает кавычки из строк и выводит на экран их содержимое. Она предназначена для удобства пользователя. Эта функция любезно добавляет пробел между каждым выводимым объектом, а также символ новой строки в конце:

```
>>> print(99, 'bottles', 'would be enough.')
99 bottles would be enough.
```

Если вам не нужны пробелы или переход на новую строку, вскоре вы узнаете, как избежать их появления.

Интерпретатор выводит строку с одинарными кавычками и *управляющими символами* вроде `\n`, что объясняется в подразделе «Создаем управляющие символы с помощью символа \» далее в текущем разделе.

Наконец, вам может понадобиться работать с *пустой строкой*. В ней нет символов, но она совершенно корректна. Вы можете создать пустую строку с помощью любых упомянутых ранее кавычек:

```
>>> ''
''
>>> ""
''
>>> ''''''
''
>>> """"""
''
>>>
```

Зачем может понадобиться пустая строка? Иногда приходится компоновать строку из других строк и для этого нужно начать с чистого листа, то есть с пустой строки.

```
>>> bottles = 99
>>> base = ''
>>> base += 'current inventory: '
>>> base += str(bottles)
>>> base
'current inventory: 99'
```

Преобразование типов данных с помощью функции str()

Вы можете преобразовывать другие типы данных Python в строки с помощью функции `str()`:

```
>>> str(98.6)
'98.6'
>>> str(1.0e4)
'10000.0'
>>> str(True)
'True'
```

В Python функция `str()` также используется для внутренних нужд, когда вы вызываете функцию `print()` для объектов, которые не являются строками, и при выполнении *интерполяции строк*, с которой вы познакомитесь в главе 7.

Создаем управляющие символы с помощью символа \

Python позволяет вам создавать *управляющие последовательности* внутри строк, чтобы добиться эффекта, который по-другому было бы трудно выразить. Размещая перед символом обратный слеш (\), вы наделяете этот символ особым значением. Наиболее распространена последовательность \n, которая означает переход на новую строку. С ее помощью вы можете создать многострочные строки из однострочных:

```
>>> palindrome = 'A man,\nA plan,\nA canal:\nPanama.'
>>> print(palindrome)
A man,
A plan,
A canal:
Panama.
```

Вы также увидите последовательность \t (табуляция), которая используется для выравнивания текста:

```
>>> print('\tabc')
    abc
>>> print('a\tbc')
a    bc
>>> print('ab\tc')
ab      c
>>> print('abc\t')
abc
```

В последней строке табуляция стоит в конце, ее вы, конечно, увидеть не можете.

Кроме того, вам могут понадобиться последовательности \' или \", чтобы поместить в строку одинарные или двойные кавычки, которые окружены таким же символом:

```
>>> print('\tabc')
    abc
>>> print('a\tbc')
a    bc
>>> print('ab\tc')
ab      c
>>> print('abc\t')
abc
```

А если вам нужен обратный слеш, просто напечатайте два:

```
>>> speech = 'Today we honor our friend, the backslash: \\.'
>>> print(speech)
Today we honor our friend, the backslash: \.
```

Объединяем строки с помощью символа +

Вы можете объединить строки или строковые переменные в Python с помощью оператора +, как показано далее:

```
>>> 'Release the kraken! ' + 'At once!'
'Release the kraken! At once!'
```

Можно также объединять строки (не переменные), просто расположив одну перед другой:

```
>>> "My word! " "A gentleman caller!"
'My word! A gentleman caller!'
```

Python не добавляет пробелы за вас при конкатенации строк, поэтому в предыдущем примере нужно явно добавить пробелы. Далее мы добавляем пробелы между каждым аргументом выражения `print()`, а также символ новой строки в конце:

```
>>> a = 'Duck.'
>>> b = a
>>> c = 'Grey Duck!'
>>> a + b + c
'Duck.Duck.Grey Duck!'
```

Размножаем строки с помощью символа *

Оператор * можно использовать для того, чтобы размножить строку. Попробуйте ввести в интерактивный интерпретатор следующие строки и посмотреть, что получится:

```
>>> start = 'Na ' * 4 + '\n'
>>> middle = 'Hey ' * 3 + '\n'
>>> end = 'Goodbye.'
>>> print(start + start + middle + end)
```

Извлекаем символ с помощью символов [ ]

Для того чтобы получить один символ строки, задайте *смещение* внутри квадратных скобок после имени строки. Смещение первого (крайнего слева) символа равно 0, следующего — 1 и т. д. Смещение последнего (крайнего справа) символа может быть выражено как –1, поэтому вам не придется считать, в таком случае смещение последующих символов будет равно –2, –3 и т. д.:

```
>>> letters = 'abcdefghijklmnopqrstuvwxyz'
>>> letters[0]
'a'
>>> letters[1]
'b'
>>> letters[-1]
'z'
```

```
>>> letters[-2]
'y'
>>> letters[25]
'z'
>>> letters[5]
'f'
```

Если вы укажете смещение, равное длине строки или больше (помните, смещения лежат в диапазоне от 0 до длины строки −1), сгенерируется исключение:

```
>>> letters[100]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
```

Индексирование работает и для других типов последовательностей (списков и кортежей), которые мы рассмотрим в главе 3.

Поскольку строки неизменяемы, вы не можете вставить символ непосредственно в строку или изменить символ по заданному индексу. Попробуем изменить слово Henny на слово Penny и посмотрим, что произойдет:

```
>>> name = 'Henny'
>>> name[0] = 'P'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

Вместо этого вам придется использовать комбинацию строковых функций вроде `replace()` или `slice` (ее вы увидите совсем скоро):

```
>>> name = 'Henny'
>>> name.replace('H', 'P')
'Penny'
>>> 'P' + name[1:]
'Penny'
```

Извлекаем подстроки с помощью оператора [start : end : step]

Из строки можно извлечь *подстроку* (часть строки) с помощью функции `slice`. Вы определяете `slice` с помощью квадратных скобок, смещения начала подстроки *start* и конца подстроки *end*, а также опционального размера шага *step*. Некоторые из этих параметров могут быть исключены. В подстроку будут включены символы, расположенные начиная с точки, на которую указывает смещение *start*, и заканчивая точкой, на которую указывает смещение *end*.

- Оператор `[:]` извлекает всю последовательность от начала до конца.
- Оператор `[start :]` извлекает последовательность с точки, на которую указывает смещение *start*, до конца.

- Оператор [: *end*] извлекает последовательность от начала до точки, на которую указывает смещение *end* минус 1.
- Оператор [*start* : *end*] извлекает последовательность с точки, на которую указывает смещение *start*, до точки, на которую указывает смещение *end* минус 1.
- Оператор [*start* : *end* : *step*] извлекает последовательность с точки, на которую указывает смещение *start*, до точки, на которую указывает смещение *end* минус 1, опуская символы, чье смещение внутри подстроки кратно *step*.

Как и ранее, смещение слева направо определяется как 0, 1 и т. д., а справа налево — как -1, -2 и т. д. Если вы не укажете смещение *start*, функция будет использовать в качестве его значения 0 (начало строки). Если вы не укажете смещение *end*, функция будет использовать конец строки.

Создадим строку, содержащую английские буквы в нижнем регистре:

```
>>> letters = 'abcdefghijklmnopqrstuvwxyz'
```

Использование простого двоеточия аналогично использованию последовательности 0: (целая строка):

```
>>> letters[:]
'abcdefghijklmnopqrstuvwxyz'
```

Вот так можно получить все символы, начиная с 20-го и заканчивая последним:

```
>>> letters[20:]
'uvwxyz'
```

А теперь — начиная с 10-го и заканчивая последним:

```
>>> letters[10:]
'klmnopqrstuvwxyz'
```

А теперь получим символы с 12-го по 14-й (Python не включает символ, расположенный под номером, который указан последним):

```
>>> letters[12:15]
'mno'
```

Последние три символа:

```
>>> letters[-3:]
'xyz'
```

В следующем примере мы начинаем со смещения 18 и идем до четвертого с конца символа. Обратите внимание на разницу с предыдущим примером, где старт с позиции -3 получал символ x. В этом примере конец диапазона -3 означает, что последним будет символ по адресу -4 — w:

```
>>> letters[18:-3]
'stuvw'
```

В следующем примере мы получаем символы, начиная с шестого с конца и заканчивая третьим с конца:

```
>>> letters[-6:-2]
'uvwx'
```

Если вы хотите увеличить шаг, укажите его после второго двоеточия, как показано в нескольких следующих примерах.

Каждый седьмой символ с начала до конца:

```
>>> letters[::7]
'ahov'
```

Каждый третий символ, начиная со смещения 4 и заканчивая 19-м символом:

```
>>> letters[4:20:3]
'ehknqt'
```

Каждый четвертый символ, начиная с 19-го:

```
>>> letters[19::4]
'tx'
```

Каждый пятый символ от начала до 20-го:

```
>>> letters[:21:5]
'afkpu'
```

Опять же значение *end* должно быть на единицу больше, чем реальное смещение.

И это еще не все! Если задать отрицательный шаг, любезный Python будет двигаться в обратную сторону. В следующем примере движение начинается с конца и заканчивается в начале, ни один символ не пропущен:

```
>>> letters[-1::-1]
'zyxwvutsrqponmlkjihgfedcba'
```

Оказывается, можно добиться того же результата, использовав такой пример:

```
>>> letters[::-1]
'zyxwvutsrqponmlkjihgfedcba'
```

Операция `slice` более мягко относится к неправильным смещениям, чем поиск по индексу. Если указать смещение меньшее, чем начало строки, оно будет обрабатываться как 0, а если указать смещение большее, чем конец строки, оно будет обработано как –1. Это показано в следующих примерах.

Начиная с –50-го символа и до конца:

```
>>> letters[-50:]
'abcdefghijklmnopqrstuvwxyz'
```

Начиная с –51-го символа и заканчивая –50-м:

```
>>> letters[-51:-50]
''
```

От начала до 69-го символа:

```
>>> letters[:70]
'abcdefghijklmnopqrstuvwxyz'
```

Начиная с 70-го символа и заканчивая 70-м:

```
>>> letters[70:71]
''
```

Получаем длину строки с помощью функции len()

До этого момента мы использовали специальные знаки препинания вроде +, чтобы манипулировать строками. Но существует не так уж много подобных функций. Теперь мы начнем использовать некоторые встроенные *функции* Python: именованные фрагменты кода, которые выполняют определенные операции.

Функция `len()` подсчитывает символы в строке:

```
>>> len(letters)
26
>>> empty = ""
>>> len(empty)
0
```

Вы можете использовать функцию `len()` и для других типов последовательностей, что показано в главе 3.

Разделяем строку с помощью функции split()

В отличие от функции `len()` некоторые функции характерны только для строк. Для того чтобы использовать строковую функцию, введите имя строки, точку, имя функции и *аргументы*, которые нужны функции: `строка.функция(аргументы)`. Более подробно о функциях мы будем говорить в разделе «Функции» главы 4.

Вы можете использовать встроенную функцию `split()`, чтобы разбить строку на *список* небольших строк, основываясь на *разделителе*. Со списками вы познакомитесь в следующей главе. Список — это последовательность значений, разделенных запятыми и окруженных квадратными скобками:

```
>>> todos = 'get gloves,get mask,give cat vitamins,call ambulance'
>>> todos.split(',')
['get gloves', 'get mask', 'give cat vitamins', 'call ambulance']
```

В предыдущем примере строка имела имя `todos`, а строковая функция называлась `split()` и получала один аргумент `','`. Если вы не укажете разделитель, функция `split()` будет использовать любую последовательность пробелов, а также символы новой строки и табуляцию:

```
>>> todos.split()
['get', 'gloves,get', 'mask,give', 'cat', 'vitamins,call', 'ambulance']
```

Если вы вызываете функцию `split` без аргументов, вам все равно нужно добавлять круглые скобки — именно так Python узнает, что вы вызываете функцию.

Объединяем строки с помощью функции join()

Для вас не должен стать великим открытием тот факт, что функция `join()` является противоположностью функции `split()`: она объединяет список строк в одну строку. Вызов функции выглядит немного запутанно, поскольку сначала вы указываете строку, которая объединяет остальные, а затем — список строк для объединения: *string*`.join(`*list*`)`. Для того чтобы объединить список строк `lines`, разделив их символами новой строки, вам нужно написать `'\n'.join(lines)`. В следующем примере мы объединим несколько имен в список, разделенный запятыми и пробелами:

```
>>> crypto_list = ['Yeti', 'Bigfoot', 'Loch Ness Monster']
>>> crypto_string = ', '.join(crypto_list)
>>> print('Found and signing book deals:', crypto_string)
Found and signing book deals: Yeti, Bigfoot, Loch Ness Monster
```

Развлекаемся со строками

Python содержит большой набор функций для работы со строками. Рассмотрим принцип работы самых распространенных из них. Объектом для тестов станет следующая строка, содержащая текст бессмертного стихотворения *What Is Liquid?* Маргарет Кэвендиш (Margaret Cavendish), графини Ньюкасл:

```
>>> poem = '''All that doth flow we cannot liquid name
Or else would fire and water be the same;
But that is liquid which is moist and wet
Fire that property can never get.
Then 'tis not cold that doth the fire put out
But 'tis the wet that makes it die, no doubt.'''
```

Для начала получим первые 13 символов (их смещения лежат в диапазоне от 0 до 12):

```
>>> poem[:13]
'All that doth'
```

Сколько символов содержит это стихотворение? (Пробелы и символы новой строки учитываются.)

```
>>> len(poem)
250
```

Начинается ли стихотворение с буквосочетания `All`?

```
>>> poem.startswith('All')
True
```

Заканчивается ли оно буквосочетанием `That's all, folks!`?

```
>>> poem.endswith('That\'s all, folks!')
False
```

Найдем смещение первого включения слова `the`:

```
>>> word = 'the'
>>> poem.find(word)
73
```

А теперь — последнего:

```
>>> poem.rfind(word)
214
```

Сколько раз встречается трехбуквенное сочетание `the`?

```
>>> poem.count(word)
3
```

Являются ли все символы стихотворения буквами или цифрами?

```
>>> poem.isalnum()
False
```

Нет, в стихотворении имеются еще и знаки препинания.

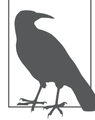
Регистр и выравнивание

В этом разделе мы рассмотрим еще несколько примеров использования встроенных функций. В качестве подопытной выберем следующую строку:

```
>>> setup = 'a duck goes into a bar...'
```

Удалим символ «.» с обоих концов строки:

```
>>> setup.strip('.')
'a duck goes into a bar'
```



Поскольку строки неизменяемы, ни один из этих примеров не изменяет строку setup. Каждый пример просто берет значение переменной setup, выполняет над ним некоторое действие, а затем возвращает результат как новую строку.

Напишем первое слово с большой буквы:

```
>>> setup.capitalize()
'A duck goes into a bar...'
```

Напишем все слова с большой буквы:

```
>>> setup.title()
'A Duck Goes Into A Bar...'
```

Запишем все слова большими буквами:

```
>>> setup.upper()
'A DUCK GOES INTO A BAR...'
```

Запишем все слова маленькими буквами:

```
>>> setup.lower()
'a duck goes into a bar...'
```

Сменим регистры букв:

```
>>> setup.swapcase()
'a DUCK GOES INTO A BAR...'
```

Теперь мы поработаем с функциями выравнивания. Строка выравнивается внутри заданного количества пробелов (в данном примере 30).

Отцентруем строку в промежутке из 30 пробелов:

```
>>> setup.center(30)
'  a duck goes into a bar...   '
```

Выровняем ее по левому краю:

```
>>> setup.ljust(30)
'a duck goes into a bar...     '
```

А теперь по правому:

```
>>> setup.rjust(30)
'     a duck goes into a bar...'
```

О форматировании и преобразовании строк мы более подробно поговорим в главе 7. Там также будет затронуто использование символа `%` и функции `format()`.

Заменяем символы с помощью функции replace()

Вы можете использовать функцию `replace()` для того, чтобы заменить одну подстроку другой. Вы передаете в эту функцию старую подстроку, новую подстроку и количество включений старой подстроки, которое нужно заменить. Если вы опустите последний аргумент, будут заменены все включения. В этом примере совпадает с заданным значением и заменяется следующая строка:

```
>>> setup.replace('duck', 'marmoset')
'a marmoset goes into a bar...'
```

Заменим максимум 100 включений:

```
>>> setup.replace('a ', 'a famous ', 100)
'a famous duck goes into a famous bar...'
```

Если вы точно знаете, какую подстроку или подстроки хотите изменить, функция `replace()` станет для вас хорошим выбором. Но будьте осторожны. Во втором при-

мере, если бы мы заменили строку из одного символа 'a', а не строку из двух символов "a " (после a идет пробел), мы бы заменили символы 'a' и в середине слов:

```
>>> setup.replace('a', 'a famous', 100)
'a famous duck goes into a famous ba famousr...'
```

Иногда вам нужно убедиться, что подстрока является целым словом, началом слова и т. д. В этих случаях понадобятся *регулярные выражения*. Они подробно описаны в главе 7.

Больше действий со строками

В Python имеется гораздо больше функций для работы со строками, чем я сейчас описал. Некоторые из них мы рассмотрим в следующих главах, но вы можете найти описания их всех в стандартной документации (http://bit.ly/py-docs-strings).

Упражнения

В этой главе были показаны атомы Python: числа, строки и переменные. Выполним несколько небольших упражнений по работе с ними с помощью интерактивного интерпретатора.

1. Сколько секунд в часе? Используйте интерактивный интерпретатор как калькулятор и умножьте количество секунд в минуте (60) на количество минут в часе (тоже 60).
2. Присвойте результат вычисления предыдущего задания (секунды в часе) переменной, которая называется seconds_per_hour.
3. Сколько секунд в сутках? Используйте переменную seconds_per_hour.
4. Снова посчитайте количество секунд в сутках, но на этот раз сохраните результат в переменной seconds_per_day.
5. Разделите значение переменной seconds_per_day на значение переменной seconds_per_hour. Используйте деление с плавающей точкой (/).
6. Разделите значение переменной seconds_per_day на значение переменной seconds_per_hour. Используйте целочисленное деление (//). Совпадает ли полученный результат с ответом на предыдущее упражнение, если не учитывать символы .0 в конце?

3 Наполнение Python: списки, кортежи, словари и множества

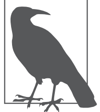
В главе 2 мы начали с базовых типов данных Python: булевых значений, целочисленных значений, чисел с плавающей точкой и строк. Если представлять их как атомы, то структуры данных, которые мы рассмотрим в этой главе, можно назвать молекулами. Так и есть: мы объединим эти базовые типы в более сложные структуры. Вы будете использовать их каждый день. Бóльшая часть работы программиста представляет собой разделение и склеивание данных в конкретные формы, поэтому сейчас вы узнаете, как пользоваться ножовками и клеевыми пистолетами.

Списки и кортежи

Большинство языков программирования могут представлять последовательность в виде объектов, проиндексированных с помощью их позиции, выраженной целым числом: первый, второй и далее до последнего. Вы уже знакомы со *строками*, они являются последовательностями символов. Вы уже немного знакомы со списками, они являются последовательностями, содержащими данные любого типа.

В Python есть еще две структуры-последовательности: *кортежи* и *списки*. Они могут содержать ноль или более элементов. В отличие от строк элементы могут быть разных типов. Фактически каждый элемент может быть *любым* объектом Python. Это позволяет создавать структуры любой сложности и глубины.

Почему же в Python имеются как списки, так и кортежи? Кортежи *неизменяемы*, когда вы присваиваете кортежу элемент, он «запекается» и больше не изменяется. Списки же *можно* изменять — добавлять и удалять элементы в любой удобный момент. Я покажу вам множество примеров применения обоих типов, сделав акцент на списках.



Вы могли слышать два возможных варианта произношения слова tuple (кортеж). Какой же из них является правильным? Если вы ответите неправильно, станут ли вас называть позером? Не волнуйтесь. Гвидо ван Россум, создатель языка Python, написал (http://bit.ly/tupletweet): «Я произношу слово tuple как too-pull по понедельникам, средам и пятницам и как tub-pull — по вторникам, четвергам и субботам. В воскресенье я вообще о них не говорю :)».

Списки

Списки служат для того, чтобы хранить объекты в определенном порядке, особенно если порядок или содержимое могут изменяться. В отличие от строк список можно изменить. Вы можете изменить список, добавить в него новые элементы, а также удалить или перезаписать существующие. Одно и то же значение может встречаться в списке несколько раз.

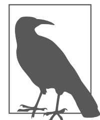
Создание списков с помощью оператора [] или метода list()

Список можно создать из нуля или более элементов, разделенных запятыми и заключенных в квадратные скобки:

```
>>> empty_list = [ ]
>>> weekdays = ['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday']
>>> big_birds = ['emu', 'ostrich', 'cassowary']
>>> first_names = ['Graham', 'John', 'Terry', 'Terry', 'Michael']
```

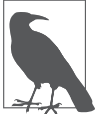
Кроме того, с помощью функции list() можно создать пустой список:

```
>>> another_empty_list = list()
>>> another_empty_list
[]
```



В разделе «Включения» главы 4 показан еще один способ создать список, который называется включением списка.

Только список weekdays использует тот факт, что элементы стоят в определенном порядке. Список first_names показывает, что значения не должны быть уникальными.



Если вы хотите размещать в последовательности только уникальные значения, множество (set) может оказаться лучшим вариантом, чем список. В предыдущем примере список big_birds вполне может быть множеством. О множествах вы можете прочесть далее в этой главе.

Преобразование других типов данных в списки с помощью функции list()

Функция list() преобразует другие типы данных в списки. В следующем примере строка преобразуется в список, состоящий из односимвольных строк:

```
>>> list('cat')
['c', 'a', 't']
```

В этом примере *кортеж* (этот тип мы рассмотрим сразу после списков) преобразуется в список:

```
>>> a_tuple = ('ready', 'fire', 'aim')
>>> list(a_tuple)
['ready', 'fire', 'aim']
```

Как я упоминал в подразделе «Разделяем строку с помощью функции split()» раздела «Строки» главы 2, можно использовать функцию `split()`, чтобы преобразовать строку в список, указав некую строку-разделитель:

```
>>> birthday = '1/6/1952'
>>> birthday.split('/')
['1', '6', '1952']
```

Что, если в оригинальной строке содержится несколько включений строки-разделителя подряд? В этом случае в качестве элемента списка вы получите пустую строку:

```
>>> splitme = 'a/b//c/d///e'
>>> splitme.split('/')
['a', 'b', '', 'c', 'd', '', '', 'e']
```

Если бы вы использовали разделитель `//`, состоящий из двух символов, то получили бы следующий результат:

```
>>> splitme = 'a/b//c/d///e'
>>> splitme.split('//')
>>>
['a/b', 'c/d', '/e']
```

Получение элемента с помощью конструкции [смещение]

Как и для строк, вы можете извлечь одно значение из списка, указав его смещение:

```
>>> marxes = ['Groucho', 'Chico', 'Harpo']
>>> marxes[0]
'Groucho'
>>> marxes[1]
'Chico'
>>> marxes[2]
'Harpo'
```

Опять же, как и в случае со строками, отрицательные индексы отсчитываются с конца строки:

```
>>> marxes[-1]
'Harpo'
>>> marxes[-2]
```

```
'Chico'
>>> marxes[-3]
'Groucho'
>>>
```

Смещение должно быть корректным значением для списка — оно представляет собой позицию, на которой располагается присвоенное ранее значение. Если вы укажете позицию, которая находится перед списком или после него, будет сгенерировано исключение (ошибка). Вот что случится, если мы попробуем получить шестого брата Маркс (Marxes) (смещение равно 5, если считать от нуля) или же пятого перед списком:

```
>>> marxes = ['Groucho', 'Chico', 'Harpo']
>>> marxes[5]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
>>> marxes[-5]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

Списки списков

Списки могут содержать элементы различных типов, включая другие списки, что показано далее:

```
>>> small_birds = ['hummingbird', 'finch']
>>> extinct_birds = ['dodo', 'passenger pigeon', 'Norwegian Blue']
>>> carol_birds = [3, 'French hens', 2, 'turtledoves']
>>> all_birds = [small_birds, extinct_birds, 'macaw', carol_birds]
```

Как же будет выглядеть список списков `all_birds`?

```
>>> all_birds
[['hummingbird', 'finch'], ['dodo', 'passenger pigeon', 'Norwegian Blue'], 'macaw',
[3, 'French hens', 2, 'turtledoves']]
```

Взглянем на его первый элемент:

```
>>> all_birds[0]
['hummingbird', 'finch']
```

Первый элемент является списком — это список `small_birds`, он указан как первый элемент списка `all_birds`. Вы можете догадаться, чем является второй элемент:

```
>>> all_birds[1]
['dodo', 'passenger pigeon', 'Norwegian Blue']
```

Это второй указанный нами элемент, `extinct_birds`. Если нужно получить первый элемент списка `extinct_birds`, мы можем извлечь его из списка `all_birds`, указав два индекса:

```
>>> all_birds[1][0]
'dodo'
```

Индекс `[1]` ссылается на второй элемент списка `all_birds`, а `[0]` — на первый элемент внутреннего списка.

Изменение элемента с помощью конструкции [смещение]

По аналогии с получением значения списка с помощью его смещения вы можете изменить это значение:

```
>>> marxes = ['Groucho', 'Chico', 'Harpo']
>>> marxes[2] = 'Wanda'
>>> marxes
['Groucho', 'Chico', 'Wanda']
```

Опять же смещение должно быть корректным для заданного списка.

Вы не можете изменить таким способом символ в строке, поскольку строки неизменяемы. Списки же можно изменить. Можете изменить количество элементов в списке, а также сами элементы.

Отрежьте кусочек — извлечение элементов с помощью диапазона смещений

Можно извлечь из списка подпоследовательность, использовав *разделение списка*:

```
>>> marxes = ['Groucho', 'Chico,' 'Harpo']
>>> marxes[0:2]
['Groucho', 'Chico']
```

Такой фрагмент списка также является списком.

Как и в случае со строками, при разделении можно пропускать некоторые значения. В следующем примере мы извлечем каждый нечетный элемент:

```
>>> marxes[::2]
['Groucho', 'Harpo']
```

Теперь начнем с последнего элемента и будем смещаться влево на 2:

```
>>> marxes[::-2]
['Harpo', 'Groucho']
```

И наконец, рассмотрим прием инверсии списка:

```
>>> marxes[::-1]
['Harpo', 'Chico', 'Groucho']
```

Добавление элемента в конец списка с помощью метода append()

Традиционный способ добавления элементов в список — вызов метода append(), чтобы добавить их в конец списка. В предыдущих примерах мы забыли о Zeppo, но ничего страшного не случилось, поскольку список можно изменить. Добавим его прямо сейчас:

```
>>> marxes.append('Zeppo')
>>> marxes
['Groucho', 'Chico', 'Harpo', 'Zeppo']
```

Объединяем списки с помощью метода extend() или оператора +=

Можно объединить один список с другим с помощью метода extend(). Предположим, что добрый человек дал нам новый список братьев Маркс, который называется others, и мы хотим добавить его в основной список marxes:

```
>>> marxes = ['Groucho', 'Chico', 'Harpo', 'Zeppo']
>>> others = ['Gummo', 'Karl']
>>> marxes.extend(others)
>>> marxes
['Groucho', 'Chico', 'Harpo', 'Zeppo', 'Gummo', 'Karl']
```

Можно также использовать оператор +=:

```
>>> marxes = ['Groucho', 'Chico', 'Harpo', 'Zeppo']
>>> others = ['Gummo', 'Karl']
>>> marxes += others
>>> marxes
['Groucho', 'Chico', 'Harpo', 'Zeppo', 'Gummo', 'Karl']
```

Если бы мы использовали метод append(), список others был бы добавлен как *один* элемент списка, вместо того чтобы объединить его элементы со списком marxes:

```
>>> marxes = ['Groucho', 'Chico', 'Harpo', 'Zeppo']
>>> others = ['Gummo', 'Karl']
>>> marxes.append(others)
>>> marxes
['Groucho', 'Chico', 'Harpo', 'Zeppo', ['Gummo', 'Karl']]
```

Это снова демонстрирует, что список может содержать элементы разных типов. В этом случае — четыре строки и список из двух строк.

Добавление элемента с помощью функции insert()

Функция append() добавляет элементы только в конец списка. Когда вам нужно добавить элемент в заданную позицию, используйте функцию insert(). Если вы укажете позицию 0, элемент будет добавлен в начало списка. Если позиция находится за пределами списка, элемент будет добавлен в конец списка, как и в случае с функцией append(), поэтому вам не нужно беспокоиться о том, что Python сгенерирует исключение:

```
>>> marxes.insert(3, 'Gummo')
>>> marxes
['Groucho', 'Chico', 'Harpo', 'Gummo', 'Zeppo']
>>> marxes.insert(10, 'Karl')
>>> marxes
['Groucho', 'Chico', 'Harpo', 'Gummo', 'Zeppo', 'Karl']
```

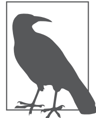
Удаление заданного элемента с помощью функции del

Наши консультанты только что проинформировали нас о том, что Гуммо (Gummo) был одним из братьев Маркс, а Карл (Karl) — не был. Отменим последний ввод:

```
>>> del marxes[-1]
>>> marxes
['Groucho', 'Chico', 'Harpo', 'Gummo', 'Zeppo']
```

Когда вы удаляете заданный элемент, все остальные элементы, которые идут следом за ним, смещаются, чтобы занять место удаленного элемента, а длина списка уменьшается на единицу. Если вы удалите 'Harpo' из последней версии списка, то получите такой результат:

```
>>> marxes = ['Groucho', 'Chico', 'Harpo', 'Gummo', 'Zeppo']
>>> marxes[2]
'Harpo'
>>> del marxes[2]
>>> marxes
['Groucho', 'Chico', 'Gummo', 'Zeppo']
>>> marxes[2]
'Gummo'
```



del является оператором Python, а не методом списка — нельзя написать marxes[-2].del(). Он похож на противоположную присваиванию (=) операцию: открепляет имя от объекта Python и может освободить память объекта, если это имя являлось последней ссылкой на нее.

Удаление элемента по значению с помощью функции remove()

Если вы не знаете точно или вам все равно, в какой позиции находится элемент, используйте функцию `remove()`, чтобы удалить его по значению. Прощай, Гуммо:

```
>>> marxes = ['Groucho', 'Chico', 'Harpo', 'Gummo', 'Zeppo']
>>> marxes.remove('Gummo')
>>> marxes
['Groucho', 'Chico', 'Harpo', 'Zeppo']
```

Получение заданного элемента и его удаление с помощью функции pop()

Вы можете получить элемент из списка и в то же время удалить его с помощью функции `pop()`. Если вызовете функцию `pop()` и укажете некоторое смещение, она возвратит элемент, находящийся в заданной позиции; если аргумент не указан, будет использовано значение –1. Так, вызов `pop(0)` вернет головной (начальный) элемент списка, а вызов `pop()` или `pop(-1)` — хвостовой (конечный) элемент, как показано далее:

```
>>> marxes = ['Groucho', 'Chico', 'Harpo', 'Zeppo']
>>> marxes.pop()
'Zeppo'
>>> marxes
['Groucho', 'Chico', 'Harpo']
>>> marxes.pop(1)
'Chico'
>>> marxes
['Groucho', 'Harpo']
```

Пришло время компьютерного жаргона! Не волнуйтесь, этого не будет на итоговом экзамене. Если вы используете функцию append(), чтобы добавить новые элементы в конец списка, и функцию pop(), чтобы удалить из конца этого же списка, вы реализуете структуру данных, известную как LIFO (last in, first out — «последним пришел — первым ушел»). Такую структуру чаще называют стеком. Вызов pop(0) создаст очередь FIFO (first in first out — «первым пришел — первым ушел»). Эти структуры могут оказаться полезными, если вы хотите собирать данные по мере их поступления и работать либо с самыми старыми (FIFO), либо с самыми новыми (LIFO).

Определение смещения элемента по значению с помощью функции index()

Если вы хотите узнать смещение элемента в списке по его значению, используйте функцию `index()`:

```
>>> marxes = ['Groucho', 'Chico', 'Harpo', 'Zeppo']
>>> marxes.index('Chico')
1
```

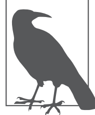
Проверка на наличие элемента в списке с помощью оператора in

В Python наличие элемента в списке проверяется с помощью оператора `in`:

```
>>> marxes = ['Groucho', 'Chico', 'Harpo', 'Zeppo']
>>> 'Groucho' in marxes
True
>>> 'Bob' in marxes
False
```

Одно и то же значение может встретиться больше одного раза. До тех пор пока оно находится в списке хотя бы в единственном экземпляре, оператор `in` будет возвращать значение `True`:

```
>>> words = ['a', 'deer', 'a' 'female', 'deer']
>>> 'deer' in words
True
```



Если вы часто проверяете наличие элемента в списке и вас не волнует порядок элементов, то для хранения и поиска уникальных значений гораздо лучше подойдет множество. О множествах мы поговорим чуть позже в этой главе.

Определяем количество включений значения с помощью функции count()

Чтобы определить, сколько раз какое-либо значение встречается в списке, используйте функцию `count()`:

```
>>> marxes = ['Groucho', 'Chico', 'Harpo']
>>> marxes.count('Harpo')
1
>>> marxes.count('Bob')
0
>>> snl_skit = ['cheeseburger', 'cheeseburger', 'cheeseburger']
>>> snl_skit.count('cheeseburger')
3
```

Преобразование списка в строку с помощью функции join()

В подразделе «Объединяем строки с помощью функции join()» раздела «Строки» главы 2 функция `join()` рассматривается более подробно, но взгляните еще на один пример того, что можно сделать с ее помощью:

```
>>> marxes = ['Groucho', 'Chico', 'Harpo']
>>> ', '.join(marxes)
'Groucho, Chico, Harpo'
```

Но погодите, вам может показаться, что нужно делать все наоборот. Функция `join()` предназначается для строк, а не для списков. Вы не можете написать `marxes.join(', ')`, несмотря на то что интуитивно это кажется правильным. Аргументом для функции `join()` является эта строка или любая итерабельная последовательность строк, включая список, и она возвращает строку. Если бы функция `join()` была только методом списка, вы не смогли бы использовать ее для других итерабельных объектов вроде кортежей и строк. Если вы хотите, чтобы она работала с любым итерабельным типом, нужно написать особый код для каждого типа, чтобы обработать объединение. Будет полезно запомнить: `join()` *противоположна* `split()`, как показано здесь:

```
>>> friends = ['Harry', 'Hermione', 'Ron']
>>> separator = ' * '
>>> joined = separator.join(friends)
>>> joined
'Harry * Hermione * Ron'
>>> separated = joined.split(separator)
>>> separated
['Harry', 'Hermione', 'Ron']
>>> separated == friends
True
```

Меняем порядок элементов с помощью функции sort()

Вам часто нужно будет изменять порядок элементов по их значениям, а не по смещениям. Для этого Python предоставляет две функции:

- функцию списка `sort()`, которая сортирует *сам список*;
- общую функцию `sorted()`, которая возвращает отсортированную *копию списка*.

Если элементы списка являются числами, они по умолчанию сортируются по возрастанию. Если они являются строками, то сортируются в алфавитном порядке:

```
>>> marxes = ['Groucho', 'Chico', 'Harpo']
>>> sorted_marxes = sorted(marxes)
>>> sorted_marxes
['Chico', 'Groucho', 'Harpo']
```

`sorted_marxes` — это копия, ее создание не изменило оригинальный список:

```
>>> marxes
['Groucho', 'Chico', 'Harpo']
```

Но вызов функции списка `sort()` для `marxes` изменит этот список:

```
>>> marxes.sort()
>>> marxes
['Chico', 'Groucho', 'Harpo']
```

Если все элементы вашего списка одного типа (в списке marxes находятся только строки), функция sort() отработает корректно. Иногда можно даже смешать типы — например, целые числа и числа с плавающей точкой, — поскольку в рамках выражений они конвертируются автоматически:

```
>>> numbers = [2, 1, 4.0, 3]
>>> numbers.sort()
>>> numbers
[1, 2, 3, 4.0]
```

По умолчанию список сортируется по возрастанию, но вы можете добавить аргумент reverse=True, чтобы отсортировать список по убыванию:

```
>>> numbers = [2, 1, 4.0, 3]
>>> numbers.sort(reverse=True)
>>> numbers
[4.0, 3, 2, 1]
```

Получение длины списка с помощью функции len()

Функция len() возвращает количество элементов списка:

```
>>> marxes = ['Groucho', 'Chico', 'Harpo']
>>> len(marxes)
3
```

Присваивание с помощью оператора =, копирование с помощью функции copy()

Если вы присваиваете один список более чем одной переменной, изменение списка в одном месте повлечет за собой его изменение в остальных, как показано далее:

```
>>> a = [1, 2, 3]
>>> a
[1, 2, 3]
>>> b = a
>>> b
[1, 2, 3]
>>> a[0] = 'surprise'
>>> a
['surprise', 2, 3]
```

Что же находится в переменной b? Ее значение все еще равно [1, 2, 3] или изменилось на ['surprise', 2, 3]? Проверим:

```
>>> b
['surprise', 2, 3]
```

Помните аналогию со стикерами в главе 2? b просто ссылается на тот же список объектов, что и a, поэтому, независимо от того, с помощью какого имени мы изменяем содержимое списка, изменение отразится на обеих переменных:

```
>>> b
['surprise', 2, 3]
>>> b[0] = 'I hate surprises'
>>> b
['I hate surprises', 2, 3]
>>> a
['I hate surprises', 2, 3]
```

Вы можете *скопировать* значения в независимый новый список с помощью одного из следующих методов:

- функции copy();
- функции преобразования list();
- разбиения списка [:].

Оригинальный список снова будет присвоен переменной a. Мы создадим b с помощью функции списка copy(), c — с помощью функции преобразования list(), а d — с помощью разбиения списка:

```
>>> a = [1, 2, 3]
>>> b = a.copy()
>>> c = list(a)
>>> d = a[:]
```

Опять же b, c и d являются *копиями* a — это новые объекты, имеющие свои значения, не связанные с оригинальным списком объектов [1, 2, 3], на который ссылается a. Изменение a *не* повлияет на копии b, c и d:

```
>>> a[0] = 'integer lists are boring'
>>> a
['integer lists are boring', 2, 3]
>>> b
[1, 2, 3]
>>> c
[1, 2, 3]
>>> d
[1, 2, 3]
```

Кортежи

Кортежи, как и списки, являются последовательностями произвольных элементов. В отличие от списков кортежи *неизменяемы*. Это означает, что вы не можете добавить, удалить или изменить элементы кортежа после того, как определите его. Поэтому кортеж аналогичен константному списку.

Создание кортежей с помощью оператора ()

Синтаксис создания кортежей несколько необычен, как мы увидим в следующих примерах.

Начнем с создания пустого кортежа с помощью оператора `()`:

```
>>> empty_tuple = ()
>>> empty_tuple
()
```

Чтобы создать кортеж, содержащий один элемент или более, ставьте после каждого элемента запятую. Это вариант для кортежей с одним элементом:

```
>>> one_marx = 'Groucho',
>>> one_marx
('Groucho',)
```

Если в вашем кортеже более одного элемента, ставьте запятую после каждого из них, кроме последнего:

```
>>> marx_tuple = 'Groucho', 'Chico', 'Harpo'
>>> marx_tuple
('Groucho', 'Chico', 'Harpo')
```

При отображении кортежа Python выводит на экран скобки. Вам они совсем не нужны — кортеж определяется запятыми, — однако не повредят. Вы можете окружить ими значения, что позволяет сделать кортежи более заметными:

```
>>> marx_tuple = ('Groucho', 'Chico', 'Harpo')
>>> marx_tuple
('Groucho', 'Chico', 'Harpo')
```

Кортежи позволяют вам присвоить несколько переменных за один раз:

```
>>> marx_tuple = ('Groucho', 'Chico', 'Harpo')
>>> a, b, c = marx_tuple
>>> a
'Groucho'
>>> b
'Chico'
>>> c
'Harpo'
```

Иногда это называется *распаковкой кортежа*.

Вы можете использовать кортежи, чтобы обменять значения с помощью одного выражения, без применения временной переменной:

```
>>> password = 'swordfish'
>>> icecream = 'tuttifrutti'
```

```
>>> password, icecream = icecream, password
>>> password
'tuttifrutti'
>>> icecream
'swordfish'
>>>
```

Функция преобразования `tuple()` создает кортежи из других объектов:

```
>>> marx_list = ['Groucho', 'Chico', 'Harpo']
>>> tuple(marx_list)
('Groucho', 'Chico', 'Harpo')
```

Кортежи против списков

Вы можете использовать кортежи вместо списков, но они имеют меньше возможностей — у них нет функций `append()`, `insert()` и т. д., поскольку кортеж не может быть изменен после создания. Почему же не применять везде списки вместо кортежей?

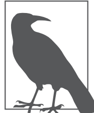
- Кортежи занимают меньше места.
- Вы не сможете уничтожить элементы кортежа по ошибке.
- Вы можете использовать кортежи как ключи словаря (см. следующий раздел).
- *Именованные кортежи* (см. пункт «Именованные кортежи» раздела «Когда лучше использовать классы и объекты, а когда — модули» главы 6) могут служить более простой альтернативой объектам.
- Аргументы функции передаются как кортежи (см. раздел «Функции» главы 4).

Более детально рассматривать кортежи я не буду. При решении повседневных задач вы будете чаще использовать списки и словари. И это хороший повод перейти к следующей теме.

Словари

Словарь очень похож на список, но порядок элементов в нем не имеет значения, и они выбираются не с помощью смещения. Вместо этого для каждого значения вы указываете связанный с ним уникальный *ключ*. Таким ключом в основном служит строка, но он может быть объектом одного из неизменяемых типов: булевой переменной, целым числом, числом с плавающей точкой, кортежем, строкой и другими объектами, которые вы увидите далее. Словари можно изменять, что означает, что вы можете добавить, удалить и изменить их элементы, которые имеют вид «ключ — значение».

Если вы работали с другими языками программирования, которые поддерживают только массивы и списки, то полюбите словари.



В других языках программирования словари могут называться ассоциативными массивами, хешами или хеш-таблицей. В Python словарь также называется dict для экономии места.

Создание словаря с помощью {}

Чтобы создать словарь, вам нужно обернуть в фигурные скобки ({}) разделенные запятыми пары `ключ : значение`. Самым простым словарем является пустой словарь, не содержащий ни ключей, ни значений:

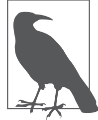
```
>>> empty_dict = {}
>>> empty_dict
{}
```

Создадим небольшой словарь, включающий цитаты из *«Словаря сатаны»* Амброза Бирса:

```
>>> bierce = {
...     "day": "A period of twenty-four hours, mostly misspent",
...     "positive": "Mistaken at the top of one's voice",
...     "misfortune": "The kind of fortune that never misses",
...     }
>>>
```

Ввод имени словаря в интерактивный интерпретатор выведет все его ключи и значения:

```
>>> bierce
{'misfortune': 'The kind of fortune that never misses',
'positive': "Mistaken at the top of one's voice",
'day': 'A period of twenty-four hours, mostly misspent'}
```



В Python допускается наличие запятой после последнего элемента списка, кортежа или словаря. Вам также не обязательно использовать выделение пробелами, как я это сделал в предыдущем примере, когда вы вводите ключи и значения внутри фигурных скобок. Это лишь улучшает читабельность.

Преобразование с помощью функции dict()

Вы можете использовать функцию `dict()`, чтобы преобразовывать последовательности из двух значений в словари. (Вы иногда можете столкнуться с последовательностями «ключ — значение» вида «Стронций, 90, углерод, 14» или «Vikings, 20, Packers, 7».) Первый элемент каждой последовательности применяется как ключ, а второй — как значение.

Для начала рассмотрим небольшой пример, использующий `lol` (список, содержащий списки, которые состоят из двух элементов):

```
>>> lol = [ ['a', 'b'], ['c', 'd'], ['e', 'f'] ]
>>> dict(lol)
{'c': 'd', 'a': 'b', 'e': 'f'}
```

Помните, что порядок ключей в словаре может быть произвольным, он зависит от того, как вы добавляете элементы.

Мы могли бы использовать любую последовательность, содержащую последовательности, которые состоят из двух элементов. Рассмотрим остальные примеры.

Список, содержащий двухэлементные кортежи:

```
>>> lot = [ ('a', 'b'), ('c', 'd'), ('e', 'f') ]
>>> dict(lot)
{'c': 'd', 'a': 'b', 'e': 'f'}
```

Кортеж, включающий двухэлементные списки:

```
>>> tol = (['a', 'b'], ['c', 'd'], ['e', 'f'])
>>> dict(tol)
{'c': 'd', 'a': 'b', 'e': 'f'}
```

Список, содержащий двухсимвольные строки:

```
>>> los = [ 'ab', 'cd', 'ef' ]
>>> dict(los)
{'c': 'd', 'a': 'b', 'e': 'f'}
```

Кортеж, содержащий двухсимвольные строки:

```
>>> tos = ('ab', 'cd', 'ef')
>>> dict(tos)
{'c': 'd', 'a': 'b', 'e': 'f'}
```

В подразделе «Итерирование по нескольким последовательностям с помощью функции zip()» раздела «Выполняем итерации с помощью for» главы 4 вы познакомитесь с функцией, которая называется `zip()`. Она позволит вам легко создавать такие двухэлементные последовательности.

Добавление или изменение элемента с помощью конструкции [ключ]

Добавить элемент в словарь довольно легко. Нужно просто обратиться к элементу по его ключу и присвоить ему значение. Если ключ уже существует в словаре, имеющееся значение будет заменено новым. Если ключ новый, он и указанное значение будут добавлены в словарь. Здесь, в отличие от списков, вам не нужно волноваться

о том, что Python сгенерирует исключение во время присваивания нового элемента, если вы укажете, что этот индекс находится вне существующего диапазона.

Создадим словарь, содержащий большинство членов Monty Python, используя их фамилии в качестве ключей, а имена — в качестве значений:

```
>>> pythons = {
...     'Chapman': 'Graham',
...     'Cleese': 'John',
...     'Idle': 'Eric',
...     'Jones': 'Terry',
...     'Palin': 'Michael',
...     }
>>> pythons
{'Cleese': 'John', 'Jones': 'Terry', 'Palin': 'Michael',
'Chapman': 'Graham', 'Idle': 'Eric'}
```

Здесь не хватает одного участника — уроженца Америки Терри Гиллиама. Перед вами попытка анонимного программиста добавить его, однако он ошибся, когда вводил имя:

```
>>> pythons['Gilliam'] = 'Gerry'
>>> pythons
{'Cleese': 'John', 'Gilliam': 'Gerry', 'Palin': 'Michael',
'Chapman': 'Graham', 'Idle': 'Eric', 'Jones': 'Terry'}
```

А вот код другого программиста, который исправил эту ошибку:

```
>>> pythons['Gilliam'] = 'Terry'
>>> pythons
{'Cleese': 'John', 'Gilliam': 'Terry', 'Palin': 'Michael',
'Chapman': 'Graham', 'Idle': 'Eric', 'Jones': 'Terry'}
```

Используя один и тот же ключ (`'Gilliam'`), мы заменили исходное значение `'Gerry'` на `'Terry'`.

Помните, что ключи в словаре должны быть *уникальными*. Именно поэтому мы в качестве ключей использовали фамилии, а не имена — двух участников Monty Python зовут Терри. Если вы применяете ключ более одного раза, победит последнее значение:

```
>>> some_pythons = {
...     'Graham': 'Chapman',
...     'John': 'Cleese',
...     'Eric': 'Idle',
...     'Terry': 'Gilliam',
...     'Michael': 'Palin',
...     'Terry': 'Jones',
...     }
>>> some_pythons
{'Terry': 'Jones', 'Eric': 'Idle', 'Graham': 'Chapman',
'John': 'Cleese', 'Michael': 'Palin'}
```

Сначала мы присвоили значение 'Gilliam' ключу 'Terry', а затем заменили его на 'Jones'.

Объединение словарей с помощью функции update()

Вы можете использовать функцию update(), чтобы скопировать ключи и значения из одного словаря в другой.

Определим словарь pythons, содержащий имена всех участников:

```
>>> pythons = {
...     'Chapman': 'Graham',
...     'Cleese': 'John',
...     'Gilliam': 'Terry',
...     'Idle': 'Eric',
...     'Jones': 'Terry',
...     'Palin': 'Michael',
...     }
>>> pythons
{'Cleese': 'John', 'Gilliam': 'Terry', 'Palin': 'Michael',
'Chapman': 'Graham', 'Idle': 'Eric', 'Jones': 'Terry'}
```

Кроме того, у нас есть другой словарь, содержащий имена других юмористов и называющийся others:

```
>>> others = { 'Marx': 'Groucho', 'Howard': 'Moe' }
```

Теперь появляется еще один анонимный программист, который считает, что члены словаря others должны быть членами Monty Python:

```
>>> pythons.update(others)
>>> pythons
{'Cleese': 'John', 'Howard': 'Moe', 'Gilliam': 'Terry',
'Palin': 'Michael', 'Marx': 'Groucho', 'Chapman': 'Graham',
'Idle': 'Eric', 'Jones': 'Terry'}
```

Что произойдет, если во втором словаре будут находиться такие же ключи, что и в первом? Победит значение из второго словаря:

```
>>> first = {'a': 1, 'b': 2}
>>> second = {'b': 'platypus'}
>>> first.update(second)
>>> first
{'b': 'platypus', 'a': 1}
```

Удаление элементов по их ключу с помощью del

Код нашего анонимного программиста был корректным — технически. Но он не должен был его писать! Члены словаря others, несмотря на свою известность и чувство

юмора, не участвовали в Monty Python. Отменим добавление последних двух элементов:

```
>>> del pythons['Marx']
>>> pythons
{'Cleese': 'John', 'Howard': 'Moe', 'Gilliam': 'Terry',
'Palin': 'Michael', 'Chapman': 'Graham', 'Idle': 'Eric',
'Jones': 'Terry'}
>>> del pythons['Howard']
>>> pythons
{'Cleese': 'John', 'Gilliam': 'Terry', 'Palin': 'Michael',
'Chapman': 'Graham', 'Idle': 'Eric', 'Jones': 'Terry'}
```

Удаление всех элементов с помощью функции clear()

Чтобы удалить все ключи и значения из словаря, вам следует использовать функцию `clear()` или просто присвоить пустой словарь заданному имени:

```
>>> pythons.clear()
>>> pythons
{}
>>> pythons = {}
>>> pythons
{}
```

Проверяем на наличие ключа с помощью in

Если вы хотите узнать, содержится ли в словаре какой-то ключ, используйте ключевое слово `in`. Снова определим словарь `pythons`, но на этот раз опустим одного-двух участников:

```
>>> pythons = {'Chapman': 'Graham', 'Cleese': 'John',
'Jones': 'Terry', 'Palin': 'Michael'}
```

Теперь проверим, кого мы добавили:

```
>>> 'Chapman' in pythons
True
>>> 'Palin' in pythons
True
```

Вспомнили ли мы о Терри Гиллиаме на этот раз?

```
>>> 'Gilliam' in pythons
False
```

Черт.

Получение элемента словаря с помощью конструкции [ключ]

Этот вариант использования словаря — самый распространенный. Вы указываете словарь и ключ, чтобы получить соответствующее значение:

```
>>> pythons['Cleese']
'John'
```

Если ключа в словаре нет, будет сгенерировано исключение:

```
>>> pythons['Marx']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'Marx'
```

Есть два хороших способа избежать возникновения этого исключения. Первый из них — проверить, имеется ли заданный ключ, с помощью ключевого слова in, что вы уже видели в предыдущем разделе:

```
>>> 'Marx' in pythons
False
```

Второй способ — использовать специальную функцию словаря get(). Вы указываете словарь, ключ и опциональное значение. Если ключ существует, вы получите связанное с ним значение:

```
>>> pythons.get('Cleese')
'John'
```

Если такого ключа нет, получите опциональное значение, если указывали его:

```
>>> pythons.get('Marx', 'Not a Python')
'Not a Python'
```

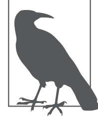
В противном случае будет возвращен объект None (интерактивный интерпретатор не выведет ничего):

```
>>> pythons.get('Marx')
>>>
```

Получение всех ключей с помощью функции keys()

Вы можете использовать функцию keys(), чтобы получить все ключи словаря. Для следующих примеров мы берем другой словарь:

```
>>> signals = {'green': 'go', 'yellow': 'go faster', 'red': 'smile for the camera'}
>>> signals.keys()
dict_keys(['green', 'red', 'yellow'])
```



В Python 2 функция keys() возвращает простой список. В Python 3 эта функция возвращает dict_keys() — итерабельное представление ключей. Это удобно для крупных словарей, поскольку не требует времени и памяти для создания и сохранения списка, которым вы, возможно, даже не воспользуетесь. Но зачастую вам нужен именно список. В Python 3 надо вызвать функцию list(), чтобы преобразовать dict_keys в список:

```
>>> list(signals.keys())
['green', 'red', 'yellow']
```

В Python 3 вам также понадобится использовать функцию list(), чтобы преобразовать результат работы функций values() и items() в обычные списки. Я пользуюсь этой функцией в своих примерах.

Получение всех значений с помощью функции values()

Чтобы получить все значения словаря, используйте функцию `values()`:

```
>>> list(signals.values())
['go', 'smile for the camera', 'go faster']
```

Получение всех пар «ключ — значение» с помощью функции items()

Когда вам нужно получить все пары «ключ — значение» из словаря, используйте функцию `items()`:

```
>>> list(signals.items())
[('green', 'go'), ('red', 'smile for the camera'), ('yellow', 'go faster')]
```

Каждая пара будет возвращена как кортеж вроде `('green', 'go')`.

Присваиваем значения с помощью оператора =, копируем их с помощью функции copy()

Как и в случае со списками, если вам нужно внести в словарь изменение, оно отразится для всех имен, которые ссылаются на него.

```
>>> signals = {'green': 'go', 'yellow': 'go faster', 'red': 'smile for the camera'}
>>> save_signals = signals
>>> signals['blue'] = 'confuse everyone'
>>> save_signals
{'blue': 'confuse everyone', 'green': 'go',
'red': 'smile for the camera', 'yellow': 'go faster'}
```

Чтобы скопировать ключи и значения из одного словаря в другой и избежать этого, вы можете воспользоваться функцией `copy()`:

```
>>> signals = {'green': 'go', 'yellow': 'go faster', 'red': 'smile for the camera'}
>>> original_signals = signals.copy()
>>> signals['blue'] = 'confuse everyone'
>>> signals
{'blue': 'confuse everyone', 'green': 'go',
'red': 'smile for the camera', 'yellow': 'go faster'}
>>> original_signals
{'green': 'go', 'red': 'smile for the camera', 'yellow': 'go faster'}
```

Множества

Множество похоже на словарь, значения которого опущены. Он имеет только ключи. Как и в случае со словарем, ключи должны быть уникальны. Если вам нужно прикрепить к ключу некую информацию, воспользуйтесь словарем.

Раньше кое-где теорию множеств преподавали в начальной школе наряду с основами математики. Если в вашей школе такого не было (или было, но вы в это время смотрели в окно, как и я), на рис. 3.1 можете увидеть объединения и пересечения множеств.

Предположим, вы хотите объединить два множества, которые содержат несколько общих ключей. Поскольку множество должно содержать только уникальные значения, объединение двух множеств будет содержать лишь одно включение каждого ключа. *Пустое* множество — это множество, содержащее ноль элементов. Основываясь на рис. 3.1, можно сказать, что пустым будет множество, которое содержит женские имена, начинающиеся с буквы «X».

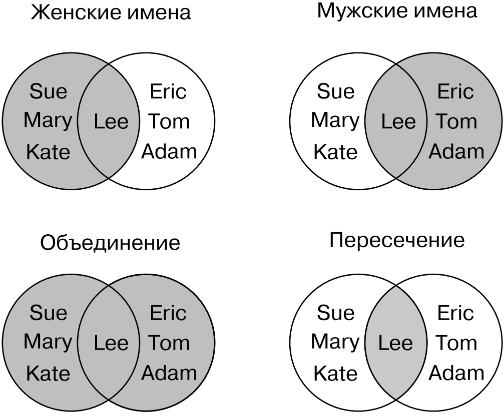


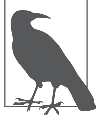
Рис. 3.1. Распространенные операции с множествами

Создание множества с помощью функции set()

Чтобы создать множество, вам следует использовать функцию `set()` или разместить в фигурных скобках одно или несколько разделенных запятыми значений, как показано здесь:

```
>>> empty_set = set()
>>> empty_set
set()
>>> even_numbers = {0, 2, 4, 6, 8}
>>> even_numbers
{0, 8, 2, 4, 6}
>>> odd_numbers = {1, 3, 5, 7, 9}
>>> odd_numbers
{9, 3, 1, 5, 7}
```

Как и в случае со словарем, порядок ключей в множестве не имеет значения.



Поскольку пустые квадратные скобки [] создают пустой список, вы могли бы рассчитывать на то, что пустые фигурные скобки {} создают пустое множество. Вместо этого пустые фигурные скобки создают пустой словарь. Именно поэтому интерпретатор выводит пустое множество как set() вместо {}. Почему так происходит? Словари появились в Python раньше и успели захватить фигурные скобки в свое распоряжение.

Преобразование других типов данных с помощью функции set()

Вы можете создать множество из списка, строки, кортежа или словаря, потеряв все повторяющиеся значения.

Для начала взглянем на строку, которая содержит более чем одно включение некоторых букв:

```
>>> set('letters')
{'l', 'e', 't', 'r', 's'}
```

Обратите внимание на то, что множество содержит только одно включение букв «e» или «t», несмотря на то, что в слове `letters` по два включения каждой из них.

Создадим множество из списка:

```
>>> set(['Dasher', 'Dancer', 'Prancer', 'Mason-Dixon'])
{'Dancer', 'Dasher', 'Prancer', 'Mason-Dixon'}
```

А теперь из кортежа:

```
>>> set(('Ummagumma', 'Echoes', 'Atom Heart Mother'))
{'Ummagumma', 'Atom Heart Mother', 'Echoes'}
```

Когда вы передаете функции set() словарь, она возвращает только ключи:

```
>>> set({'apple': 'red', 'orange': 'orange', 'cherry': 'red'})
{'apple', 'cherry', 'orange'}
```

Проверяем на наличие значения с помощью ключевого слова in

Такое использование множеств самое распространенное. Мы создадим словарь, который называется drinks. Каждый ключ будет названием коктейля, а соответствующие значения — множествами ингредиентов:

```
>>> drinks = {
...     'martini': {'vodka', 'vermouth'},
...     'black russian': {'vodka', 'kahlua'},
...     'white russian': {'cream', 'kahlua', 'vodka'},
...     'manhattan': {'rye', 'vermouth', 'bitters'},
...     'screwdriver': {'orange juice', 'vodka'}
...     }
```

Несмотря на то что и словарь, и множества окружены фигурными скобками ({ и }), множество — это всего лишь последовательность значений, а словарь — это набор пар «ключ — значение».

Какой из коктейлей содержит в себе водку? (Обратите внимание на то, что для выполнения этих проверок я заранее демонстрирую использование ключевых слов for, if, and и or, которые будут рассмотрены только в следующей главе.)

```
>>> for name, contents in drinks.items():
...     if 'vodka' in contents:
...         print(name)
...
screwdriver
martini
black russian
white russian
```

Мы хотим выпить коктейль с водкой, но не переносим лактозу, а вермут на вкус напоминает керосин:

```
>>> for name, contents in drinks.items():
...     if 'vodka' in contents and not ('vermouth' in contents or
...         'cream' in contents):
...         print(name)
...
screwdriver
black russian
```

Перепишем этот пример чуть более сжато в следующем разделе.

Комбинации и операторы

Что, если вам нужно проверить наличие сразу нескольких значений множества? Предположим, вы хотите найти любой напиток, содержащий апельсиновый сок или вермут. Для этого мы используем *оператор пересечения множеств* (&):

```
>>> for name, contents in drinks.items():
...     if contents & {'vermouth', 'orange juice'}:
...         print(name)
...
screwdriver
martini
manhattan
```

Результатом работы оператора & является множество, содержащее все элементы, которые находятся в обоих сравниваемых списках. Если ни один из заданных ингредиентов не содержится в предлагаемых коктейлях, оператор & вернет пустое множество. Этот результат можно считать равным False.

Теперь перепишем пример из предыдущего раздела, в котором мы хотели водки, не смешанной со сливками или вермутом:

```
>>> for name, contents in drinks.items():
...     if 'vodka' in contents and not contents & {'vermouth', 'cream'}:
...         print(name)
...
screwdriver
black russian
```

Сохраним множества ингредиентов для этих двух напитков в переменных, чтобы нам не пришлось набирать много текста в дальнейших примерах:

```
>>> bruss = drinks['black russian']
>>> wruss = drinks['white russian']
```

В следующих примерах демонстрируется использование операторов множеств. В одних из них демонстрируется применение особой пунктуации, в других — особых функций, в третьих — и того и другого. Мы будем использовать тестовые множества a (содержит элементы 1 и 2) и b (содержит элементы 2 и 3):

```
>>> a = {1, 2}
>>> b = {2, 3}
```

Пересечение множеств (члены обоих множеств) можно получить с помощью особого пунктуационного символа & или функции множества intersection(), как показано здесь:

```
>>> a & b
{2}
>>> a.intersection(b)
{2}
```

В этом фрагменте используются сохраненные нами переменные:

```
>>> bruss & wruss
{'kahlua', 'vodka'}
```

В этом примере мы получаем *объединение* (члены обоих множеств), используя оператор | или функцию множества `union()`:

```
>>> a | b
{1, 2, 3}
>>> a.union(b)
{1, 2, 3}
```

Алкогольная версия:

```
>>> bruss | wruss
{'cream', 'kahlua', 'vodka'}
```

Разность множеств (члены только первого множества, но не второго) можно получить с помощью символа - или функции `difference()`:

```
>>> a – b
{1}
>>> a.difference(b)
{1}
>>> bruss – wruss
set()
>>> wruss – bruss
{'cream'}
```

Самыми распространенными операциями с множествами являются объединение, пересечение и разность. Для полноты картины я включил в этот раздел и остальные операции, но вы, возможно, никогда не будете их использовать.

Для выполнения *исключающего ИЛИ* (элементы или первого, или второго множества, но не общие) используйте оператор ^ или функцию `symmetric_difference()`:

```
>>> a ^ b
{1, 3}
>>> a.symmetric_difference(b)
{1, 3}
```

В этом примере определяется эксклюзивный ингредиент для русских напитков:

```
>>> bruss ^ wruss
{'cream'}
```

Вы можете проверить, является ли одно множество *подмножеством* другого (все члены первого множества являются членами второго), с помощью оператора <= или функции `issubset()`:

```
>>> a <= b
False
>>> a.issubset(b)
False
```

Добавление сливок в коктейль «черный русский» сделает его «белым русским», поэтому wruss является подмножеством bruss:

```
>>> bruss <= wruss
True
```

Является ли любое множество подмножеством самого себя? Ага.

```
>>> a <= a
True
>>> a.issubset(a)
True
```

Для того чтобы стать *собственным подмножеством*, второе множество должно содержать все члены первого и несколько других. Определяется это с помощью оператора <:

```
>>> a < b
False
>>> a < a
False
>>> bruss < wruss
True
```

Множество множеств противоположно подмножеству (все члены второго множества являются также членами первого). Для определения этого используется оператор >= или функция issuperset():

```
>>> a >= b
False
>>> a.issuperset(b)
False
>>> wruss >= bruss
True
```

Любое множество является множеством множеств самого себя:

```
>>> a >= a
True
>>> a.issuperset(a)
True
```

И наконец, вы можете найти *собственное множество множеств* (первое множество содержит все члены второго и несколько других) с помощью оператора >:

```
>>> a > b
False
>>> wruss > bruss
True
```

Множество не может быть собственным множеством множеств самого себя:

```
>>> a > a
False
```

Сравнение структур данных

Напомню, список создается с помощью квадратных скобок ([]), кортеж — с помощью запятых, а словарь — с помощью фигурных скобок ({}). Во всех случаях вы получаете доступ к отдельному элементу с помощью квадратных скобок:

```
>>> marx_list = ['Groucho', 'Chico', 'Harpo']
>>> marx_tuple = 'Groucho', 'Chico', 'Harpo'
>>> marx_dict = {'Groucho': 'banjo', 'Chico': 'piano', 'Harpo': 'harp'}
>>> marx_list[2]
'Harpo'
>>> marx_tuple[2]
'Harpo'
>>> marx_dict['Harpo']
'harp'
```

Для списка и кортежа значение, находящееся в квадратных скобках, является целочисленным смещением. Для словаря же оно является ключом. Для всех троих результатом будет значение.

Создание крупных структур данных

Ранее мы работали с простыми булевыми значениями, числами и строками. Теперь же мы работаем со списками, кортежами, множествами и словарями. Вы можете объединить эти встроенные структуры данных в собственные структуры, более крупные и сложные. Начнем с трех разных списков:

```
>>> marxes = ['Groucho', 'Chico', 'Harpo']
>>> pythons = ['Chapman', 'Cleese', 'Gilliam', 'Jones', 'Palin']
>>> stooges = ['Moe', 'Curly', 'Larry']
```

Мы можем создать кортеж, который содержит в качестве элементов каждый из этих списков:

```
>>> tuple_of_lists = marxes, pythons, stooges
>>> tuple_of_lists
(['Groucho', 'Chico', 'Harpo'],
['Chapman', 'Cleese', 'Gilliam', 'Jones', 'Palin'],
['Moe', 'Curly', 'Larry'])
```

Можем также создать список, который содержит три списка:

```
>>> list_of_lists = [marxes, pythons, stooges]
>>> list_of_lists
[['Groucho', 'Chico', 'Harpo'],
['Chapman', 'Cleese', 'Gilliam', 'Jones', 'Palin'],
['Moe', 'Curly', 'Larry']]
```

Наконец, создадим словарь из списков. В этом примере используем название группы комиков в качестве ключа, а список ее членов — в качестве значения:

```
dict_of_lists = {'Marxes': marxes, 'Pythons': pythons, 'Stooges': stooges}
>> dict_of_lists
{'Stooges': ['Moe', 'Curly', 'Larry'],
'Marxes': ['Groucho', 'Chico', 'Harpo'],
'Pythons': ['Chapman', 'Cleese', 'Gilliam', 'Jones', 'Palin']}
```

Вас ограничивают только сами типы данных. Например, ключи словаря должны быть неизменяемыми, поэтому список, словарь или множество не могут быть ключом для другого словаря. Но кортеж может быть ключом. Например, вы можете создать алфавитный указатель достопримечательностей, основываясь на GPS-координатах (широте, долготе и высоте; обратитесь к разделу «Карты» приложения Б, где вы сможете найти еще несколько примеров работы с картами):

```
>>> houses = {
        (44.79, -93.14, 285): 'My House',
        (38.89, -77.03, 13): 'The White House'
        }
```

Упражнения

В этой главе вы познакомились с более сложными структурами данных: списками, кортежами, словарями и множествами. Используя их и типы данных, описанные в главе 2 (числа и строки), вы можете представить множество элементов реального мира.

1. Создайте список `years_list`, содержащий год, в который вы родились, и каждый последующий год вплоть до вашего пятого дня рождения. Например, если вы родились в 1980 году, список будет выглядеть так: `years_list = [1980, 1981, 1982, 1983, 1984, 1985]`.

   Если вам меньше пяти лет и вы уже читаете эту книгу, то я даже не знаю, что сказать.

2. В какой из годов, содержащихся в списке `years_list`, был ваш третий день рождения? Помните, в первый год вам было 0 лет.
3. В какой из годов, перечисленных в списке `years_list`, вам было больше всего лет?
4. Создайте список `things`, содержащий три элемента: `"mozzarella"`, `"cinderella"`, `"salmonella"`.
5. Напишите с большой буквы тот элемент списка `things`, который относится к человеку, а затем выведите список. Изменился ли элемент списка?
6. Переведите сырный элемент списка `things` в верхний регистр целиком и выведите список.

7. Удалите болезнь из списка `things`, получите Нобелевскую премию и затем выведите список на экран.
8. Создайте список, который называется `surprise` и содержит элементы `'Groucho'`, `'Chico'` и `'Harpo'`.
9. Напишите последний элемент списка `surprise` со строчной буквы, затем обратите его и напишите с прописной буквы.
10. Создайте англо-французский словарь, который называется `e2f`, и выведите его на экран. Вот ваши первые слова: dog/chien, cat/chat и walrus/morse.
11. Используя словарь `e2f`, выведите французский вариант слова walrus.
12. Создайте французско-английский словарь `f2e` на основе словаря `e2f`. Используйте метод `items`.
13. Используя словарь `f2e`, выведите английский вариант слова chien.
14. Создайте и выведите на экран множество английских слов из ключей словаря `e2f`.
15. Создайте многоуровневый словарь `life`. Используйте следующие строки для ключей верхнего уровня: `'animals'`, `'plants'` и `'other'`. Сделайте так, чтобы ключ `'animals'` ссылался на другой словарь, имеющий ключи `'cats'`, `'octopi'` и `'emus'`. Сделайте так, чтобы ключ `'cats'` ссылался на список строк со значениями `'Henri'`, `'Grumpy'` и `'Lucy'`. Остальные ключи должны ссылаться на пустые словари.
16. Выведите на экран высокоуровневые ключи словаря `life`.
17. Выведите на экран ключи `life['animals']`.
18. Выведите значения `life['animals']['cats']`.

4 Корочка Python: структуры кода

В первых трех главах вы увидели множество примеров данных, но практически не работали с ними. В большинстве примеров использовался интерактивный интерпретатор, а сами они были довольно короткими. Теперь вы увидите, как структурировать *код* Python, а не только данные.

В большинстве языков программирования символы вроде фигурных скобок ({ и }) или ключевые слова вроде `begin` и `end` применяется для того, чтобы разбить код на разделы. В этих языках хорошим тоном является использование отбивки пробелами, чтобы сделать программу более удобочитаемой для себя и других. Существуют даже инструменты, которые помогут красиво выстроить ваш код.

Гвидо ван Россум при разработке Python решил, что выделения пробелами будет достаточно, чтобы задать структуру программы и избежать ввода всех этих скобок. Python отличается от других языков тем, что *пробелы* в нем используются для того, чтобы задать структуру программы. Этот аспект новички замечают одним из первых, и он может показаться странным для тех, кто уже работал с другими языками программирования. Однако по прошествии некоторого времени это начинает казаться естественным и вы перестаете это замечать. Вы даже привыкнете к тому, что делаете больше, набирая меньше текста.

Комментируем с помощью символа

Комментарий — это фрагмент текста в вашей программе, который будет проигнорирован интерпретатором Python. Вы можете использовать комментарии, чтобы дать пояснение близлежащего кода, сделать какие-то пометки для себя, или для чего-то еще. Комментарий помечается символом #; все, что находится после # до конца текущей строки, является комментарием. Обычно комментарий располагается на отдельной строке, как показано здесь:

```
>>> # 60 с/мин * 60 мин/ч * 24 ч/день
>>> seconds_per_day = 86400
```

Или на той же строке, что и код, который нужно пояснить:

```
>>> seconds_per_day = 86400 # 60 sec/min * 60 min/hr * 24 hr/day
```

Символ # имеет много имен: *хеш*, *шарп*, *фунт* или устрашающее *октоторп*[1]. Как бы вы его ни назвали[2], его эффект действует только до конца строки, на которой он располагается.

Python не дает возможности написать многострочный комментарий. Вам нужно явно начинать каждую строку или раздел комментария с символа #:

```
>>> # Я могу сказать здесь все, даже если Python это не нравится,
... # поскольку я защищен крутым
... # октоторпом.
...
>>>
```

Однако если октоторп находится внутри текстовой строки, он становится простым символом #:

```
>>> print("No comment: quotes make the # harmless.")
No comment: quotes make the # harmless.
```

Продлеваем строки с помощью символа \

Любая программа становится более удобочитаемой, если ее строки относительно короткие. Рекомендуемая (но не обязательная) максимальная длина строки равна 80 символам. Если вы не можете выразить свою мысль в рамках 80 символов, воспользуйтесь символом *возобновления* `\`. Просто поместите его в конце строки, и дальше Python будет действовать так, будто это все та же строка.

Например, если бы я хотел создать длинную строку из нескольких коротких, я мог бы сделать это пошагово:

```
>>> alphabet = ''
>>> alphabet += 'abcdefg'
>>> alphabet += 'hijklmnop'
>>> alphabet += 'qrstuv'
>>> alphabet += 'wxyz'
```

Или же за одно действие, использовав символ `continuation`:

```
>>> alphabet = 'abcdefg' + \
...      'hijklmnop' + \
...      'qrstuv' + \
...      'wxyz'
```

Продлить строку может быть необходимо, если выражение располагается на нескольких строках:

```
>>> 1 + 2 +
  File "<stdin>", line 1
```

[1] Прямо как та восьминогая зеленая штука, которая стоит прямо за вами.
[2] Пожалуйста, не зовите его. Оно может вернуться.

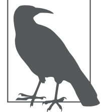
```
    1 + 2 +
          ^
SyntaxError: invalid syntax
>>> 1 + 2 + \
... 3
6
>>>
```

Сравниваем выражения с помощью операторов if, elif и else

До этого момента мы говорили только о структурах данных. Теперь же наконец готовы сделать первый шаг к рассмотрению *структур кода*, которые вводят данные в программы. (Вы уже могли получить представление о них в главе 3, в разделе о множествах.) В качестве первого примера рассмотрим небольшую программу, которая проверяет значение булевой переменной `disaster` и выводит подходящий комментарий:

```
>>> disaster = True
>>> if disaster:
...     print("Woe!")
... else:
...     print("Whee!")
...
Woe!
>>>
```

Строки `if` и `else` в Python являются *операторами*, которые проверяют, является ли значение выражения (в данном случае переменной `disaster`) равным `True`. Помните, `print()` — это встроенная в Python *функция* для вывода информации, как правило, на ваш экран.



Если вы работали с другими языками программирования, обратите внимание на то, что при проверке if вам не нужно ставить скобки. Не нужно писать что-то вроде if (disaster == True). В конце строки следует поставить двоеточие (:). Если вы, как и я, иногда забываете ставить двоеточие, Python выведет сообщение об ошибке.

Каждая строка `print()` отделена пробелами под соответствующей проверкой. Я использовал четыре пробела для того, чтобы выделить каждый подраздел. Хотя вы можете использовать любое количество пробелов, Python ожидает, что внутри одного раздела будет применяться одинаковое количество пробелов. Рекомендованный стиль — PEP-8 (http://bit.ly/pep-8) — предписывает использовать четыре

пробела. Не применяйте табуляцию или сочетание табуляций и пробелов — это мешает подсчитывать отступы.

Все выполненные в этом примере действия я объясню более детально далее в текущей главе.

1. Присвоили булево значение `True` переменной `disaster`.
2. Произвели *условное сравнение* с помощью операторов `if` и `else`, выполняя разные фрагменты кода в зависимости от значений переменной `disaster`.
3. Вызвали функцию `print()`, чтобы вывести текст на экран.

Можно организовывать проверку в проверке столько раз, сколько вам нужно:

```
>>> furry = True
>>> small = True
>>> if furry:
...     if small:
...         print("It's a cat.")
...     else:
...         print("It's a bear!")
... else:
...     if small:
...         print("It's a skink!")
...     else:
...         print("It's a human. Or a hairless bear.")
...
It's a cat.
```

В Python отступы определяют, какие разделы `if` и `else` объединены в пару. Наша первая проверка обращалась к переменной `furry`. Поскольку ее значение равно `True`, Python переходит к выделенной таким же количеством пробелов проверке `if small`. Поскольку мы указали значение переменной `small` равным `True`, проверка вернет результат `True`. Это заставит Python вывести на экран строку **It's a cat**.

Если необходимо проверить более двух вариантов, используйте операторы `if`, `elif` (это значит else if — «иначе если») и `else`:

```
>>> color = "puce"
>>> if color == "red":
...     print("It's a tomato")
... elif color == "green":
...     print("It's a green pepper")
... elif color == "bee purple":
...     print("I don't know what it is, but only bees can see it")
... else:
...     print("I've never heard of the color", color)
...
I've never heard of the color puce
```

В предыдущем примере мы проверяли равенство с помощью оператора ==. В Python используются следующие операторы сравнения:

- равенство (==);
- неравенство (!=);
- меньше (<);
- меньше или равно (<=);
- больше (>);
- больше или равно (>=);
- включение (in …).

Эти операторы возвращают булевы значения True или False. Взглянем на то, как они работают, но сначала присвоим значение переменной x:

```
>>> x = 7
```

Теперь выполним несколько проверок:

```
>>> x == 5
False
>>> x == 7
True
>>> 5 < x
True
>>> x < 10
True
```

Обратите внимание на то, что для *проверки на равенство* используются два знака «равно» (==); помните, что один знак «равно» применяется для присваивания значения переменной.

Если вам нужно выполнить несколько сравнений одновременно, можете использовать *булевы операторы* and, or и not, чтобы определить итоговый двоичный результат.

Булевы операторы имеют более низкий *приоритет*, нежели фрагменты кода, которые они сравнивают. Это значит, что сначала высчитывается результат фрагментов, а затем они сравниваются. В данном примере из-за того, что мы устанавливаем значение x равным 7, проверка 5 < x возвращает значение True и проверка x < 10 также возвращает True, поэтому наше выражение преобразуется в True and True:

```
>>> 5 < x and x < 10
True
```

Как указывается в подразделе «Приоритет операций» раздела «Числа» главы 2, самый простой способ избежать путаницы — использовать круглые скобки:

```
>>> (5 < x) and (x < 10)
True
```

Рассмотрим некоторые другие проверки:

```
>>> 5 < x or x < 10
True
>>> 5 < x and x > 10
False
>>> 5 < x and not x > 10
True
```

Если вы используете оператор and для того, чтобы объединить несколько проверок, Python позволит вам сделать следующее:

```
>>> 5 < x < 10
True
```

Это выражение аналогично проверкам 5 < x и x < 10. Вы также можете писать более длинные сравнения:

```
>>> 5 < x < 10 < 999
True
```

Что есть истина? Что, если элемент, который мы проверяем, не является булевым? Чем Python считает True и False?

Значение false не обязательно явно означает False. Например, к False приравниваются все следующие значения:

- булева переменная False;
- значение None;
- целое число 0;
- число с плавающей точкой 0.0;
- пустая строка (' ');
- пустой список ([]);
- пустой кортеж (());
- пустой словарь ({});
- пустое множество (set()).

Все остальные значения приравниваются к True. Программы, написанные на Python, используют это определение истинности (или, как в данном случае, ложности), чтобы выполнять проверку на пустоту структуры данных наряду с проверкой на равенство непосредственно значению False:

```
>>> some_list = []
>>> if some_list:
...     print("There's something in here")
... else:
...     print("Hey, it's empty!")
...
Hey, it's empty!
```

Если вы выполняете проверку для выражения, а не для простой переменной, Python оценит его значение и вернет булев результат. Поэтому, если вы введете следующее:

```
if color == "red":
```

Python оценит выражение `color == "red"`. В нашем примере мы присвоили переменной `color` значение `"puce"`, поэтому значение выражения `color == "red"` равно `False` и Python перейдет к следующей проверке:

```
elif color == "green":
```

Повторяем действия с помощью while

Проверки с помощью `if`, `elif` и `else` выполняются последовательно. Иногда нам нужно выполнить какие-то операции более чем один раз. Нам нужен *цикл*, и простейшим вариантом циклов в Python является `while`. Попробуйте запустить с помощью интерактивного интерпретатора следующий пример — это простейший цикл, который выводит на экран значения от 1 до 5:

```
>>> count = 1
>>> while count <= 5:
...     print(count)
...     count += 1
...
1
2
3
4
5
>>>
```

Сначала мы присваиваем значение `1` переменной `count`. Цикл `while` сравнивает значение переменной `count` с числом `5` и продолжает работу, если значение переменной `count` меньше или равно `5`. Внутри цикла мы выводим значение переменной `count`, а затем *увеличиваем* его на `1` с помощью выражения `count += 1`. Python возвращается к верхушке цикла и снова сравнивает значение переменной `count` с числом `5`. Значение переменной `count` теперь равно `2`, поэтому содержимое цикла `while` выполняется снова и переменная `count` увеличивается до `3`.

Это продолжается до тех пор, пока переменная `count` не будет увеличена с `5` до `6` в нижней части цикла. Во время очередного возврата наверх цикла проверка `count <= 5` вернет значение `False` и цикл `while` закончится. Python перейдет к выполнению следующих строк.

Прерываем цикл с помощью break

Если вы хотите, чтобы цикл выполнялся до тех пор, пока что-то не произойдет, но вы не знаете точно, когда это событие случится, можете воспользоваться *бесконечным циклом*, содержащим оператор `break`. В этот раз мы считаем строку с клавиатуры с помощью функции `input()`, а затем выведем ее на экран, сделав первую букву прописной. Мы прервем цикл, когда будет введена строка, содержащая только букву «q»:

```
>>> while True:
...     stuff = input("String to capitalize [type q to quit]: ")
...     if stuff == "q":
...         break
...     print(stuff.capitalize())
...
String to capitalize [type q to quit]: test
Test
String to capitalize [type q to quit]: hey, it works
Hey, it works
String to capitalize [type q to quit]: q
>>>
```

Пропускаем итерации с помощью continue

Иногда вам нужно не прерывать весь цикл, а только пропустить по какой-то причине одну итерацию. Рассмотрим воображаемый пример: считаем целое число, выведем на экран его значение в квадрате, если оно четное, и пропустим его, если оно нечетное. Мы даже добавим несколько комментариев. И вновь для выхода из цикла используем строку `"q"`:

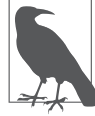
```
>>> while True:
...     value = input("Integer, please [q to quit]: ")
...     if value == 'q':      # выход
...         break
...     number = int(value)
...     if number % 2 == 0:   # нечетное число
...        continue
...     print(number, "squared is", number*number)
...
Integer, please [q to quit]: 1
1 squared is 1
Integer, please [q to quit]: 2
Integer, please [q to quit]: 3
3 squared is 9
```

```
Integer, please [q to quit]: 4
Integer, please [q to quit]: 5
5 squared is 25
Integer, please [q to quit]: q
>>>
```

Проверяем, завершился ли цикл заранее, с помощью else

Если цикл `while` завершился нормально (без вызова `break`), управление передается в опциональный блок `else`. Вы можете использовать его в цикле, где выполняете некоторую проверку и прерываете цикл, как только проверка успешно выполняется. Блок `else` выполнится в том случае, если цикл `while` будет пройден полностью, но искомый объект не будет найден:

```
>>> numbers = [1, 3, 5]
>>> position = 0
>>> while position < len(numbers):
...     number = numbers[position]
...     if number % 2 == 0:
...         print('Found even number', number)
...         break
...     position += 1
... else:  # break not called
...     print('No even number found')
...
No even number found
```



Такое использование ключевого слова else может оказаться нелогичным. Рассматривайте его как проверку на прерывание цикла.

Выполняем итерации с помощью for

В Python *итераторы* часто используются по одной простой причине. Они позволяют вам проходить структуры данных, не зная, насколько эти структуры велики и как реализованы. Вы даже можете пройти по данным, которые были созданы во время работы программы, что позволяет обработать потоки данных, которые в противном случае не поместились бы в память компьютера.

Вполне возможно пройти по последовательности таким образом:

```
>>> rabbits = ['Flopsy', 'Mopsy', 'Cottontail', 'Peter']
>>> current = 0
>>> while current < len(rabbits):
...     print(rabbits[current])
...     current += 1
```

```
...
Flopsy
Mopsy
Cottontail
Peter
```

Однако существует более характерный для Python способ решения этой задачи:

```
>>> for rabbit in rabbits:
...     print(rabbit)
...
Flopsy
Mopsy
Cottontail
Peter
```

Списки вроде `rabbits` являются одними из *итерабельных* объектов в Python наряду со строками, кортежами, словарями и некоторыми другими элементами. Итерирование по кортежу или списку возвращает один элемент за раз. Итерирование по строке возвращает один символ за раз, как показано здесь:

```
>>> word = 'cat'
>>> for letter in word:
...     print(letter)
...
c
a
t
```

Итерирование по словарю (или его функции `keys()`) возвращает ключи. В этом примере ключи являются типами карт в настольной игре Clue (за пределами Северной Америки она называется CluedoAmerica):

```
>>> accusation = {'room': 'ballroom', 'weapon': 'lead pipe',
                  'person': 'Col. Mustard'}
>>> for card in accusation:  #  или for card in accusation.keys():
...     print(card)
...
room
weapon
person
```

Чтобы итерировать по значениям, а не по ключам, следует использовать функцию `values()`:

```
>>> for value in accusation.values():
...     print(value)
...
ballroom
lead pipe
Col. Mustard
```

Чтобы вернуть как ключ, так и значение кортежа, вы можете использовать функцию `items()`:

```
>>> for item in accusation.items():
...     print(item)
...
('room', 'ballroom')
('weapon', 'lead pipe')
('person', 'Col. Mustard')
```

Помните, что можете присвоить значение кортежу за один шаг. Для каждого кортежа, возвращенного функцией `items()`, присвойте первое значение (ключ) переменной `card`, а второе (значение) — переменной `contents`:

```
>>> for card, contents in accusation.items():
...     print('Card', card, 'has the contents', contents)
...
Card weapon has the contents lead pipe
Card person has the contents Col. Mustard
Card room has the contents ballroom
```

Прерываем цикл с помощью break

Ключевое слово `break` в цикле `for` прерывает этот цикл точно так же, как и цикл `while`.

Пропускаем итерации с помощью continue

Добавление ключевого слова `continue` в цикл `for` позволяет перейти на следующую итерацию цикла, как и в случае с циклом `while`.

Проверяем, завершился ли цикл заранее, с помощью else

Как и в цикле `while`, в `for` имеется опциональный блок `else`, который проверяет, выполнился ли цикл `for` полностью. Если ключевое слово `break` **не было вызвано**, будет выполнен блок `else`.

Это полезно, если вам нужно убедиться в том, что предыдущий цикл выполнился полностью, вместо того чтобы рано прерваться. Цикл `for` в следующем примере выводит на экран название сыра и прерывается, если сыра в магазине не найдется:

```
>>> cheeses = []
>>> for cheese in cheeses:
...     print('This shop has some lovely', cheese)
...     break
```

```
... else:  # отсутствие прерывания означает, что сыра нет
...     print('This is not much of a cheese shop, is it?')
...
This is not much of a cheese shop, is it?
```

Как и в цикле while, в цикле for использование блока else может показаться нелогичным. Можно рассматривать цикл for как поиск чего-то, в таком случае else будет вызываться, если вы ничего не нашли. Чтобы получить тот же эффект без блока else, используйте переменную, которая будет показывать, нашелся ли искомый элемент в цикле for, как здесь:

```
>>> cheeses = []
>>> found_one = False
>>> for cheese in cheeses:
...     found_one = True
...     print('This shop has some lovely', cheese)
...     break
...
>>> if not found_one:
...     print('This is not much of a cheese shop, is it?')
...
This is not much of a cheese shop, is it?
```

Итерирование по нескольким последовательностям с помощью функции zip()

Существует еще один полезный прием — параллельное итерирование по нескольким последовательностям с помощью функции zip():

```
>>> days = ['Monday', 'Tuesday', 'Wednesday']
>>> fruits = ['banana', 'orange', 'peach']
>>> drinks = ['coffee', 'tea', 'beer']
>>> desserts = ['tiramisu', 'ice cream', 'pie', 'pudding']
>>> for day, fruit, drink, dessert in zip(days, fruits, drinks, desserts):
...     print(day, ": drink", drink, "eat", fruit, "enjoy", dessert)
...
Monday : drink coffee — eat banana — enjoy tiramisu
Tuesday : drink tea — eat orange — enjoy ice cream
Wednesday : drink beer — eat peach — enjoy pie
```

Функция zip() прекращает свою работу, когда выполняется самая короткая последовательность. Один из списков (desserts) оказался длиннее остальных, поэтому никто не получит пудинг, пока мы не увеличим остальные списки.

В разделе «Словари» главы 3 показывается, как с помощью функции dict() можно создавать словари из последовательностей, содержащих два элемента, вроде кортежей, списков или строк. Вы можете использовать функцию zip(), чтобы

пройти по нескольким последовательностям и создать кортежи из элементов с одинаковыми смещениями. Создадим два кортежа из соответствующих друг другу английских и французских слов:

```
>>> english = 'Monday', 'Tuesday', 'Wednesday'
>>> french = 'Lundi', 'Mardi', 'Mercredi'
```

Теперь используем функцию zip(), чтобы объединить эти кортежи в пару. Значение, возвращаемое функцией zip(), само по себе не является списком или кортежем, но его можно преобразовать в любую из этих последовательностей:

```
>>> list(zip(english, french))
[('Monday', 'Lundi'), ('Tuesday', 'Mardi'), ('Wednesday', 'Mercredi')]
```

Передайте результат работы функции zip() непосредственно функции dict() — и у нас готов небольшой англо-французский словарь!

```
>>> dict(zip(english, french))
{'Monday': 'Lundi', 'Tuesday': 'Mardi', 'Wednesday': 'Mercredi'}
```

Генерирование числовых последовательностей с помощью функции range()

Функция range() возвращает поток чисел в заданном диапазоне без необходимости создавать и сохранять крупную структуру данных вроде списка или кортежа. Это позволяет вам создавать большие диапазоны, не использовав всю память компьютера и не обрушив программу.

Вы можете применять функцию range() аналогично slice(): range(start, stop, step). Если опустите значение start, диапазон начнется с 0. Необходимым является лишь значение stop: как и в случае со slice(), оно определяет последнее значение, которое будет создано прямо перед остановкой функции. Значение по умолчанию step равно 1, но вы можете изменить его на -1.

Как и zip(), функция range() возвращает *итерабельный* объект, поэтому вам нужно пройти по значениям с помощью конструкции for … in или преобразовать объект в последовательность вроде списка. Создадим диапазон 0, 1, 2:

```
>>> for x in range(0,3):
...     print(x)
...
0
1
2
>>> list(range(0, 3))
[0, 1, 2]
```

Вот так можно создать диапазон от 2 до 0:

```
>>> for x in range(2, -1, -1):
...     print(x)
```

```
...
2
1
0
>>> list(range(2, -1, -1))
[2, 1, 0]
```

В следующем фрагменте кода используется шаг 2, чтобы получить все четные числа от 0 до 10:

```
>>> list(range(0, 11, 2))
[0, 2, 4, 6, 8, 10]
```

Прочие итераторы

В главе 8 рассматривается итерирование по файлам. В главе 6 вы сможете увидеть, как использовать итерирование для объектов, которые сами определили.

Включения

Включение — это компактный способ создать структуру данных из одного или более итераторов. Включения позволяют вам объединять циклы и условные проверки, не используя при этом громоздкий синтаксис. Если вы применяете включение, то можно сказать, что уже неплохо знаете Python. Иными словами, это одна из характерных особенностей данного языка.

Включение списков

Вы можете создать список целых чисел от 1 до 5, добавляя их туда по одному за раз, например, так:

```
>>> number_list = []
>>> number_list.append(1)
>>> number_list.append(2)
>>> number_list.append(3)
>>> number_list.append(4)
>>> number_list.append(5)
>>> number_list
[1, 2, 3, 4, 5]
```

Или же вы могли бы использовать итератор и функцию range():

```
>>> number_list = []
>>> for number in range(1, 6):
...     number_list.append(number)
...
>>> number_list
[1, 2, 3, 4, 5]
```

Или же преобразовать в список сам результат работы функции `range()`:

```
>>> number_list = list(range(1, 6))
>>> number_list
[1, 2, 3, 4, 5]
```

Все эти подходы абсолютно корректны с точки зрения Python и сгенерируют одинаковый результат. Однако более характерным для Python является создание списка с помощью *включения списка*. Простейшая форма такого включения выглядит так:

```
[выражение for элемент in итерабельный объект]
```

Вот так выглядит включение списка целых чисел:

```
>>> number_list = [number for number in range(1,6)]
>>> number_list
[1, 2, 3, 4, 5]
```

В первой строке вам нужно, чтобы первая переменная `number` сформировала значения для списка: следует разместить результат работы цикла в переменной `number_list`. Вторая переменная `number` является частью цикла `for`. Чтобы показать, что первая переменная `number` является выражением, попробуем такой вариант:

```
>>> number_list = [number-1 for number in range(1,6)]
>>> number_list
[0, 1, 2, 3, 4]
```

Включение списка перемещает цикл в квадратные скобки. Этот пример включения ненамного проще предыдущего, но это еще не все. Включение списка может содержать условное выражение, которое выглядит примерно так:

```
[выражение for элемент in итерабельный объект if условие]
```

Создадим новое включение, которое создает список, состоящий только из четных чисел, расположенных в диапазоне от 1 до 5 (помните, что выражение `number % 2` имеет значение `True` для четных чисел и `False` для нечетных):

```
>>> a_list = [number for number in range(1,6) if number % 2 == 1]
>>> a_list
[1, 3, 5]
```

Теперь включение выглядит чуть более компактно, чем его традиционный аналог:

```
>>> a_list = []
>>> for number in range(1,6):
...     if number % 2 == 1:
...         a_list.append(number)
...
>>>  a_list
[1, 3, 5]
```

Наконец, точно так же, как и в случае вложенных циклов, можно написать более чем один набор операторов `for ...` в соответствующем выделении. Чтобы продемонстрировать это, сначала создадим старый добрый вложенный цикл и выведем на экран результат:

```
>>> rows = range(1,4)
>>> cols = range(1,3)
>>> for row in rows:
...     for col in cols:
...         print(row, col)
...
1 1
1 2
2 1
2 2
3 1
3 2
```

Теперь воспользуемся включением и присвоим его переменной `cells`, создавая тем самым список кортежей `(row, col)`:

```
>>> rows = range(1,4)
>>> cols = range(1,3)
>>> cells = [(row, col) for row in rows for col in cols]
>>> for cell in cells:
...     print(cell)
...
(1, 1)
(1, 2)
(2, 1)
(2, 2)
(3, 1)
(3, 2)
```

Кстати, вы можете воспользоваться распаковкой кортежа, чтобы выдернуть значения `row` и `col` из каждого кортежа по мере итерирования по списку `cells`:

```
>>> for row, col in cells:
...     print(row, col)
...
1 1
1 2
2 1
2 2
3 1
3 2
```

Фрагменты `for row ...` и `for col ...` во включении также могут иметь свои проверки `if`.

Включение словаря

Для словарей также можно создать включение. Простейшая его форма выглядит привычно:

```
{ выражение_ключа: выражение_значения for выражение in итерабельный объект }
```

Как и в случае с включениями списка, выделения словарей также имеют проверки `if` и несколько операторов `for`:

```
>>> word = 'letters'
>>> letter_counts = {letter: word.count(letter) for letter in word}
>>> letter_counts
{'l': 1, 'e': 2, 't': 2, 'r': 1, 's': 1}
```

Мы запускаем цикл, проходя по каждой из семи букв в строке `letters`, и считаем, сколько раз появляется эта буква. Два наших вызова `word.count(letter)` — это лишь пустая трата времени, поскольку нам нужно подсчитать буквы «e» и «t» два раза. Но когда мы считаем буквы «e» во второй раз, то не причиняем вреда, поскольку лишь заменяем уже существующую запись в словаре; то же относится и к подсчету букв «t». Следующий способ решения задачи более характерен для Python:

```
>>> word = 'letters'
>>> letter_counts = {letter: word.count(letter) for letter in set(word)}
>>> letter_counts
{'t': 2, 'l': 1, 'e': 2, 'r': 1, 's': 1}
```

Ключи словаря располагаются в ином, чем в предыдущем примере, порядке, поскольку итерирование по результату работы функции `set(word)` возвращает буквы в другом порядке, нежели итерирование по строке `word`.

Включение множества

Никто не хочет оказаться обиженным, поэтому даже у множеств есть включения. Простейшая версия выглядит как включение списка или словаря, которые вы только что видели:

```
{ выражение for выражение in итерабельный объект}
```

Более длинные версии (проверки `if`, множественные операторы `for`) также доступны для множеств:

```
>>> a_set = {number for number in range(1,6) if number % 3 == 1}
>>> a_set
{1, 4}
```

Включение генератора

Для кортежей не существует включений. Вы могли подумать, что замена квадратных скобок у выделения списка на круглые создаст включение кортежа. Может даже показаться, что это работает, поскольку исключение не будет сгенерировано, если вы напишете следующее:

```
>>> number_thing = (number for number in range(1, 6))
```

В круглые скобки заключено включение генератора, оно возвращает объект генератора:

```
>>> type(number_thing)
<class 'generator'>
```

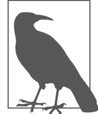
Сами генераторы мы рассмотрим более детально позже в данной главе. Применение генераторов — это один из способов предоставить данные итератору.

Вы можете итерировать непосредственно по этому объекту генератора, как показано здесь:

```
>>> for number in number_thing:
...     print(number)
...
1
2
3
4
5
```

Или же вы можете обернуть вызов `list()` вокруг включения генератора, чтобы заставить его работать как включение списка:

```
>>> number_list = list(number_thing)
>>> number_list
[1, 2, 3, 4, 5]
```



Генератор может быть запущен лишь однажды. Списки, множества и словари существуют в памяти, но генератор создает свои значения во время работы программы и выдает их по одному за раз через итератор. Он не запоминает их, поэтому вы не можете перезапустить или создать резервную копию генератора.

Если вы попробуете проитерировать по генератору заново, то обнаружите, что он истощен:

```
>>> try_again = list(number_thing)
>>> try_again
[]
```

Вы можете создать генератор из включения генератора, как мы сделали это здесь, или из *функции генератора*. Сначала мы поговорим о функциях в целом, а затем рассмотрим частный случай — функции генератора.

Функции

До этого момента все наши примеры кода представляли собой небольшие фрагменты. Они годятся для решения небольших задач, но никто не хочет набирать эти фрагменты раз за разом. Нам нужен какой-то способ организовать большой фрагмент кода в более удобные фрагменты.

Первый шаг к повторному использованию кода — это создание функций. *Функция* — это именованный фрагмент кода, отделенный от других. Она может принимает любое количество любых входных *параметров* и возвращать любое количество любых *результатов*.

С функцией можно сделать две вещи:

- *определить*;
- *вызвать*.

Чтобы определить функцию, вам нужно написать `def`, имя функции, входные параметры, заключенные в скобки, и, наконец, двоеточие (`:`). Имена функций подчиняются тем же правилам, что и имена переменных (они должны начинаться с буквы или _ и содержать только буквы, цифры или _).

Давайте действовать пошагово. Сначала определим и вызовем функцию, которая не имеет параметров. Перед вами пример простейшей функции:

```
>>> def do_nothing():
...     pass
```

Даже если функции не нужны параметры, вам все равно придется указать круглые скобки и двоеточие в ее определении. Следующую строку необходимо выделить пробелами точно так же, как если бы это был оператор `if`. Python требует использовать выражение `pass`, чтобы показать, что функция ничего не делает. Это эквивалентно утверждению «Эта страница специально оставлена пустой» (несмотря на то что теперь это не так).

Функцию можно вызвать, просто написав ее имя и скобки. Она сработает так, как я и обещал, вполне успешно не сделав ничего:

```
>>> do_nothing()
>>>
```

Теперь определим и вызовем другую функцию, которая не имеет параметров и выводит на экран одно слово:

```
>>> def make_a_sound():
...     print('quack')
...
```

```
>>> make_a_sound()
quack
```

Когда вы вызываете функцию `make_a_sound()`, Python выполняет код, расположенный внутри ее описания. В этом случае он выводит одно слово и возвращает управление основной программе.

Попробуем написать функцию, которая не имеет параметров, но возвращает значение:

```
>>> def agree():
...     return True
...
```

Вы можете вызвать эту функцию и проверить возвращаемое ею значение с помощью `if`:

```
>>> if agree():
...     print('Splendid!')
... else:
...     print('That was unexpected.')
...
Splendid!
```

Только что вы сделали большой шаг. Комбинация функций с проверками вроде `if` и циклами вроде `while` позволяет вам делать ранее недоступные вещи.

Теперь пришло время поместить что-нибудь в эти скобки. Определим функцию `echo()`, имеющую один параметр `anything`. Она использует оператор `return`, чтобы отправить значение `anything` вызывающей стороне дважды, разделив их пробелом:

```
>>> def echo(anything):
...     return anything + ' ' + anything
...
>>>
```

Теперь вызовем функцию `echo()`, передав ей строку `'Rumplestiltskin'`:

```
>>> echo('Rumplestiltskin')
'Rumplestiltskin Rumplestiltskin'
```

Значения, которые вы передаете в функцию при вызове, называются *аргументами*. Когда вы вызываете функцию с аргументами, значения этих аргументов копируются в соответствующие *параметры* внутри функций. В предыдущем примере функции `echo()` передавалась строка `'Rumplestiltskin'`. Это значение копировалось внутри функции `echo()` в параметр `anything`, а затем возвращалось (в этом случае оно удваивалось и разделялось пробелом) вызывающей стороне.

Эти примеры функций довольно просты. Напишем функцию, которая принимает аргумент и что-то с ним делает. Мы адаптируем предыдущий фрагмент кода, который комментировал цвета. Назовем его `commentary` и сделаем так, чтобы он

принимал в качестве аргумента строку color. Сделаем так, чтобы он возвращал описание строки вызывающей стороне, которая может решить, что с ним делать дальше:

```
>>> def commentary(color):
...     if color == 'red':
...         return "It's a tomato."
...     elif color == "green":
...         return "It's a green pepper."
...     elif color == 'bee purple':
...         return "I don't know what it is, but only bees can see it."
...     else:
...         return "I've never heard of the color "  + color +  "."
...
>>>
```

Вызовем функцию commentary(), передав ей в качестве аргумента строку 'blue'.

```
>>> comment = commentary('blue')
```

Функция сделает следующее:

- присвоит значение 'blue' параметру функции color;
- пройдет по логической цепочке if-elif-else;
- вернет строку;
- присвоит строку переменной comment.

Что мы получим в результате?

```
>>> print(comment)
I've never heard of the color blue.
```

Функция может принимать любое количество аргументов (включая нуль) любого типа. Она может возвращать любое количество результатов (также включая нуль) любого типа. Если функция не вызывает return явно, вызывающая сторона получит результат None.

```
>>> print(do_nothing())
None
```

None может быть полезным

None — это специальное значение в Python, которое заполняет собой пустое место, если функция ничего не возвращает. Оно не является булевым значением False, несмотря на то что похоже на него при проверке булевой переменной. Рассмотрим пример:

```
>>> thing = None
>>> if thing:
...     print("It's some thing")
```

```
... else:
...     print("It's no thing")
...
It's no thing
```

Для того чтобы понять важность отличия `None` от булева значения `False`, используйте оператор `is`:

```
>>> if thing is None:
...     print("It's nothing")
... else:
...     print("It's something")
...
It's nothing
```

Разница кажется небольшой, однако она важна в Python. `None` потребуется вам, чтобы отличить отсутствующее значение от пустого. Помните, что целочисленные нули, нули с плавающей точкой, пустые строки (`''`), списки (`[]`), кортежи (`(,)`), словари (`{}`) и множества (`set()`) все равны `False`, но не равны `None`.

Напишем небольшую функцию, которая выводит на экран проверку на равенство `None`:

```
>>> def is_none(thing):
...     if thing is None:
...         print("It's None")
...     elif thing:
...         print("It's True")
...     else:
...         print("It's False")
...
```

Теперь выполним несколько проверок:

```
>>> is_none(None)
It's None
>>> is_none(True)
It's True
>>> is_none(False)
It's False
>>> is_none(0)
It's False
>>> is_none(0.0)
It's False
>>> is_none(())
It's False
>>> is_none([])
It's False
>>> is_none({})
It's False
>>> is_none(set())
It's False
```

Позиционные аргументы

Python довольно гибко обрабатывает аргументы функций в сравнении с многими языками программирования. Наиболее распространенный тип аргументов — это *позиционные аргументы*, чьи значения копируются в соответствующие параметры согласно порядку следования.

Эта функция создает словарь из позиционных входных аргументов и возвращает его:

```
>>> def menu(wine, entree, dessert):
...     return {'wine': wine, 'entree': entree, 'dessert': dessert}
...
>>> menu('chardonnay', 'chicken', 'cake')
{'dessert': 'cake', 'wine': 'chardonnay', 'entree': 'chicken'}
```

Несмотря на распространенность аргументов такого типа, у них есть недостаток, который заключается в том, что вам нужно запоминать значение каждой позиции. Если бы мы вызвали функцию `menu()`, передав в качестве последнего аргумента марку вина, обед вышел бы совершенно другим:

```
>>> menu('beef', 'bagel', 'bordeaux')
{'dessert': 'bordeaux', 'wine': 'beef', 'entree': 'bagel'}
```

Аргументы — ключевые слова

Для того чтобы избежать путаницы с позиционными аргументами, вы можете указать аргументы с помощью имен соответствующих параметров. Порядок следования аргументов в этом случае может быть иным:

```
>>> menu(entree='beef', dessert='bagel', wine='bordeaux')
{'dessert': 'bagel', 'wine': 'bordeaux', 'entree': 'beef'}
```

Вы можете объединять позиционные аргументы и аргументы — ключевые слова. Сначала выберем вино, а для десерта и основного блюда используем аргументы — ключевые слова.

```
>>> menu('frontenac', dessert='flan', entree='fish')
{'entree': 'fish', 'dessert': 'flan', 'wine': 'frontenac'}
```

Если вы вызываете функцию, имеющую как позиционные аргументы, так и аргументы — ключевые слова, то позиционные аргументы необходимо указывать первыми.

Указываем значение параметра по умолчанию

Вы можете указать значения по умолчанию для параметров. Значения по умолчанию используются в том случае, если вызывающая сторона не предоставила соот-

ветствующий аргумент. Эта приятная особенность может оказаться довольно полезной. Воспользуемся предыдущим примером:

```
>>> def menu(wine, entree, dessert='pudding'):
...     return {'wine': wine, 'entree': entree, 'dessert': dessert}
```

В этот раз мы вызовем функцию menu(), не передав ей аргумент dessert:

```
>>> menu('chardonnay', 'chicken')
{'dessert': 'pudding', 'wine': 'chardonnay', 'entree': 'chicken'}
```

Если вы предоставите аргумент, он будет использован вместо аргумента по умолчанию:

```
>>> menu('dunkelfelder', 'duck', 'doughnut')
{'dessert': 'doughnut', 'wine': 'dunkelfelder', 'entree': 'duck'}
```

Значение аргументов по умолчанию высчитывается, когда функция определяется, а не выполняется. Распространенной ошибкой новичков (и иногда не совсем новичков) является использование изменяемого типа данных вроде списка или словаря в качестве аргумента по умолчанию.

В следующей проверке ожидается, что функция buggy() будет каждый раз запускаться с новым пустым списком result, добавлять в него аргумент arg, а затем выводить на экран список, состоящий из одного элемента. Однако в этой функции есть баг: список будет пуст только при первом вызове. Во второй раз список result будет содержать элемент, оставшийся после предыдущего вызова:

```
>>> def buggy(arg, result=[]):
...     result.append(arg)
...     print(result)
...
>>> buggy('a')
['a']
>>> buggy('b')   # ожидаем увидеть ['b']
['a', 'b']
```

Функция работала бы корректно, если бы код выглядел так:

```
>>> def works(arg):
...     result = []
...     result.append(arg)
...     return result
...
>>> works('a')
['a']
>>> works('b')
['b']
```

Решить проблему можно, передав в функцию что-то еще, чтобы указать на то, что вызов является первым:

```
>>> def nonbuggy(arg, result=None):
...     if result is None:
...         result = []
...     result.append(arg)
...     print(result)
...
>>> nonbuggy('a')
['a']
>>> nonbuggy('b')
['b']
```

Получаем позиционные аргументы с помощью *

Если вы работали с языками программирования C или C++, то можете предположить, что астериск (*) в Python как-то относится к указателям. Это не так, Python не имеет указателей.

Если символ * будет использован внутри функции с параметром, произвольное количество позиционных аргументов будет сгруппировано в кортеж. В следующем примере args является кортежем параметров, который был создан из аргументов, переданных в функцию print_args():

```
>>> def print_args(*args):
...     print('Positional argument tuple:', args)
...
```

Если вы вызовете функцию без аргументов, то получите пустой кортеж:

```
>>> print_args()
Positional argument tuple: ()
```

Все аргументы, которые вы передадите, будут выведены на экран как кортеж args:

```
>>> print_args(3, 2, 1, 'wait!', 'uh...')
Positional argument tuple: (3, 2, 1, 'wait!', 'uh...')
```

Это полезно при написании функций вроде print(), которые принимают произвольное количество аргументов. Если в вашей функции имеются также обязательные позиционные аргументы, *args отправится в конец списка и получит все остальные аргументы:

```
>>> def print_more(required1, required2, *args):
...     print('Need this one:', required1)
...     print('Need this one too:', required2)
...     print('All the rest:', args)
...
```

```
>>> print_more('cap', 'gloves', 'scarf', 'monocle', 'mustache wax')
Need this one: cap
Need this one too: gloves
All the rest: ('scarf', 'monocle', 'mustache wax')
```

При использовании * вам не нужно обязательно называть кортеж параметров `args`, однако это распространенная идиома в Python.

Получение аргументов — ключевых слов с помощью **

Вы можете использовать два астериска (**), чтобы сгруппировать аргументы — ключевые слова в словарь, где имена аргументов станут ключами, а их значения — соответствующими значениями в словаре. В следующем примере определяется функция `print_kwargs()`, в которой выводятся ее аргументы — ключевые слова:

```
>>> def print_kwargs(**kwargs):
...     print('Keyword arguments:', kwargs)
...
```

Теперь попробуйте вызвать ее, передав несколько аргументов:

```
>>> print_kwargs(wine='merlot', entree='mutton', dessert='macaroon')
Keyword arguments: {'dessert': 'macaroon', 'wine': 'merlot', 'entree': 'mutton'}
```

Внутри функции `kwargs` является словарем.

Если вы используете позиционные аргументы и аргументы — ключевые слова (`*args` и `**kwargs`), они должны следовать в этом же порядке. Как и в случае с `args`, вам не обязательно называть этот словарь `kwargs`, но это опять же является распространенной практикой.

Строки документации

Дзен Python гласит: **удобочитаемость имеет значение**. Вы можете прикрепить документацию к определению функции, включив строку в начало ее тела. Она называется строкой документации:

```
>>> def echo(anything):
...     'echo returns its input argument'
...     return anything
```

Вы можете сделать строку документации довольно длинной и даже, если хотите, применить к ней форматирование, что показано в следующем примере:

```
def print_if_true(thing, check):
    '''
    Prints the first argument if a second argument is true.
    The operation is:
```

```
        1. Check whether the *second* argument is true.
        2. If it is, print the *first* argument.
    '''
    if check:
        print(thing)
```

Для того чтобы вывести строку документации некоторой функции, вам следует вызвать функцию help(). Передайте ей имя функции, чтобы получить список всех аргументов и красиво отформатированную строку документации:

```
>>> help(echo)
Help on function echo in module __main__:
echo(anything)
    echo returns its input argument
```

Если вы хотите увидеть только строку документации без форматирования:

```
>>> print(echo.__doc__)
echo returns its input argument
```

Подозрительно выглядящая строка __doc__ является внутренним именем строки документации как переменной внутри функции. В пункте «Использование _ и __ в именах» в разделе «Пространства имен и область определения» данной главы объясняется причина появления всех этих нижних подчеркиваний.

Функции — это объекты первого класса

Я уже упоминал мантру Python: объектами является все, включая числа, строки, кортежи, списки, словари и даже функции. Функции в Python являются объектами первого класса. Вы можете присвоить их переменным, использовать как аргументы для других функций и возвращать из функций. Это дает вам возможность решать с помощью Python такие задачи, справиться с которыми средствами многих других языков сложно, если не невозможно.

Для того чтобы убедиться в этом, определим простую функцию answer(), которая не имеет аргументов и просто выводит число 42:

```
>>> def answer():
...     print(42)
```

Вы знаете, что получите в качестве результата, если запустите эту функцию:

```
>>> answer()
42
```

Теперь определим еще одну функцию с именем run_something. Она имеет один аргумент, который называется func и представляет собой функцию, которую нужно запустить. Эта функция просто вызывает другую функцию:

```
>>> def run_something(func):
...     func()
```

Если мы передадим `answer` в функцию `run_something()`, то используем ее как данные, прямо как и другие объекты:

```
>>> run_something(answer)
42
```

Обратите внимание: вы передали строку `answer`, а не `answer()`. В Python круглые скобки означают «вызови эту функцию». Если скобок нет, Python относится к функции как к любому другому объекту. Это происходит потому, что, как и все остальное в Python, функция является объектом:

```
>>> type(run_something)
<class 'function'>
```

Попробуем запустить функцию с аргументами. Определим функцию `add_args()`, которая выводит на экран сумму двух числовых аргументов, `arg1` и `arg2`:

```
>>> def add_args(arg1, arg2):
...     print(arg1 + arg2)
```

Чем является `add_args()`?

```
>>> type(add_args)
<class 'function'>
```

Теперь определим функцию, которая называется `run_something_with_args()` и принимает три аргумента:

- `func` — функция, которую нужно запустить;
- `arg1` — первый аргумент функции `func`;
- `arg2` — второй аргумент функции `func`:

  ```
  >>> def run_something_with_args(func, arg1, arg2):
  ...     func(arg1, arg2)
  ```

Когда вы вызываете функцию `run_something_with_args()`, та функция, что передается вызывающей стороной, присваивается параметру `func`, а переменные `arg1` и `arg2` получают значения, которые следуют далее в списке аргументов. Вызов `func(arg1, arg2)` выполняет данную функцию с этими аргументами, потому что круглые скобки указывают Python сделать это.

Проверим функцию `run_something_with_args()`, передав ей имя функции `add_args` и аргументы `5` и `9`:

```
>>> run_something_with_args(add_args, 5, 9)
14
```

Внутри функции `run_something_with_args()` аргумент `add_args`, представляющий собой имя функции, был присвоен параметру `func`, `5` — параметру `arg1`, а `9` — параметру `arg2`. В итоге получается следующая конструкция:

```
add_args(5, 9)
```

Вы можете объединить этот прием с использованием `*args` и `**kwargs`.

Определим тестовую функцию, которая принимает любое количество позиционных аргументов, определяет их сумму с помощью функции `sum()` и возвращает ее:

```
>>> def sum_args(*args):
...    return sum(args)
```

Я не упоминал функцию `sum()` ранее. Это встроенная в Python функция, которая высчитывает сумму значений итерабельного числового (целочисленного или с плавающей точкой) аргумента.

Мы определим новую функцию `run_with_positional_args()`, принимающую функцию и произвольное количество позиционных аргументов, которые нужно будет передать в нее:

```
>>> def run_with_positional_args(func, *args):
...    return func(*args)
```

Теперь вызовем ее:

```
>>> run_with_positional_args(sum_args, 1, 2, 3, 4)
10
```

Вы можете использовать функции как элементы списков, кортежей, множеств и словарей. Функции неизменяемы, поэтому вы можете даже применять их как ключи для словарей.

Внутренние функции

Вы можете определить функцию внутри другой функции:

```
>>> def outer(a, b):
...     def inner(c, d):
...         return c + d
...     return inner(a, b)
...
>>>
>>> outer(4, 7)
11
```

Внутренние функции могут быть полезны при выполнении некоторых сложных задач более одного раза внутри другой функции. Это позволит избежать использования циклов или дублирования кода. Рассмотрим пример работы со строкой, когда внутренняя функция добавляет текст к своему аргументу:

```
>>> def knights(saying):
...     def inner(quote):
...         return "We are the knights who say: '%s'" % quote
```

```
...     return inner(saying)
...
>>> knights('Ni!')
"We are the knights who say: 'Ni!'"
```

Замыкания

Внутренняя функция может действовать как *замыкание*. Замыкание — это функция, которая динамически генерируется другой функцией, и они обе могут изменяться и запоминать значения переменных, которые были созданы вне функции.

Следующий пример создан на основе предыдущего примера `knights()`. Назовем новую функцию `knights2()`, поскольку у нас нет воображения, и превратим функцию `inner()` в замыкание, которое называется `inner2()`. Различия заключаются в следующем.

- `inner2()` использует внешний параметр `saying` непосредственно, вместо того чтобы получить его как аргумент.
- `knights2()` возвращает имя функции `inner2`, вместо того чтобы вызывать ее:

```
>>> def knights2(saying):
...     def inner2():
...         return "We are the knights who say: '%s'" % saying
...     return inner2
...
```

Функция `inner2()` знает значение переменой `saying`, которое было передано в функцию, и запоминает его. Строка `inner2` возвращает эту особую копию функции `inner2`, но не вызывает ее. Это и есть замыкание: динамически созданная функция, которая запоминает, откуда она появилась.

Вызовем функцию `knights2()` два раза с разными аргументами:

```
>>> a = knights2('Duck')
>>> b = knights2('Hasenpfeffer')
```

О'кей, чем являются `a` и `b`?

```
>>> type(a)
<class 'function'>
>>> type(b)
<class 'function'>
```

Они являются функциями, а также замыканиями:

```
>>> a
<function knights2.<locals>.inner2 at 0x10193e158>
>>> b
<function knights2.<locals>.inner2 at 0x10193e1e0>
```

Если мы вызовем их, они запомнят значение переменной `saying`, которое было использовано, когда они были созданы функцией `knights2`:

```
>>> a()
"We are the knights who say: 'Duck'"
>>> b()
"We are the knights who say: 'Hasenpfeffer'"
```

Анонимные функции: функция lambda()

В Python *лямбда-функция* — это анонимная функция, выраженная одним выражением. Вы можете использовать ее вместо обычной маленькой функции.

Для того чтобы проиллюстрировать анонимные функции, сначала создадим пример, в котором используются обычные функции. Для начала мы определим функцию `edit_story()`. Она имеет следующие аргументы:

- `words` — список слов;
- `func` — функция, которая должна быть применена к каждому слову в списке `words`:

```
>>> def edit_story(words, func):
...     for word in words:
...         print(func(word))
```

Теперь нам нужны список слов и функция, которую требуется к ним применить. В качестве слов я возьму список звуков (гипотетических), которые мог бы издать мой кот, если бы (гипотетически) он не заметил одну из лестниц:

```
>>> stairs = ['thud', 'meow', 'thud', 'hiss']
```

Функция же запишет с большой буквы каждое слово и добавит к нему восклицательный знак, что идеально подойдет для заголовка какой-нибудь желтой кошачьей газетенки:

```
>>> def enliven(word):   # больше эмоций!
...     return word.capitalize() + '!'
```

Смешаем наши ингредиенты:

```
>>> edit_story(stairs, enliven)
Thud!
Meow!
Thud!
Hiss!
```

Наконец переходим к лямбде. Функция `enliven()` была такой короткой, что мы можем заменить ее лямбдой:

```
>>>
>>> edit_story(stairs, lambda word: word.capitalize() + '!')
Thud!
```

```
Meow!
Thud!
Hiss!
>>>
```

Лямбда принимает один аргумент, который в этом примере назван `word`. Все, что находится между двоеточием и закрывающей скобкой, является определением функции.

Часто использование настоящих функций вроде `enliven()` гораздо прозрачнее, чем использование лямбд. Лямбды наиболее полезны в случаях, когда вам нужно определить множество мелких функций и запомнить все их имена. В частности, вы можете использовать лямбды в графических пользовательских интерфейсах, чтобы определить *функции внешнего вызова*. Примеры вы можете найти в приложении А.

Генераторы

В Python *генератор* — это объект, который предназначен для создания последовательностей. С его помощью вы можете проитерировать потенциально огромные последовательности без необходимости создания и сохранения всей последовательности в память сразу. Генераторы часто становятся источником данных для итераторов. Как вы помните, мы уже использовали один из них, `range()`, в примерах кода для того, чтобы сгенерировать последовательность целых чисел. В Python 2 функция `range()` возвращает список, ограниченный так, чтобы он помещался в память. В Python 2 также есть функция `xrange()`, которая стала обычной функцией `range()` в Python 3. В этом примере складываются все целые числа от 1 до 100:

```
>>> sum(range(1, 101))
5050
```

Каждый раз, когда вы итерируете через генератор, он отслеживает, где он находился во время последнего вызова, и возвращает следующее значение. Это отличает его от обычной функции, которая не помнит о предыдущих вызовах и всегда начинает работу с первой строки и в неизменном состоянии.

Если вы хотите создать потенциально большую последовательность и ее код слишком велик для того, чтобы создать включение генератора, напишите *функцию генератора*. Это обычная функция, но она возвращает значение с помощью выражения `yield`, а не `return`. Напишем собственную функцию `range()`:

```
>>> def my_range(first=0, last=10, step=1):
...     number = first
...     while number < last:
...         yield number
...         number += step
...
```

Это нормальная функция:

```
>>> my_range
<function my_range at 0x10193e268>
```

И она возвращает объект генератора:

```
>>> ranger = my_range(1, 5)
>>> ranger
<generator object my_range at 0x101a0a168>
```

Мы можем проитерировать по этому объекту генератора:

```
>>> for x in ranger:
...     print(x)
...
1
2
3
4
```

Декораторы

Иногда вам нужно модифицировать существующую функцию, не меняя при этом ее исходный код. Зачастую нужно добавить выражение для отладки, чтобы посмотреть, какие аргументы были туда переданы.

Декоратор — это функция, которая принимает одну функцию в качестве аргумента и возвращает другую функцию. Мы используем следующие приемы из нашего арсенала:

- `*args` и `**kwargs`;
- внутренние функции;
- функции в качестве аргументов.

Функция `document_it()` определяет декоратор, который:

- выведет имя функции и значение переданных в нее аргументов;
- запустит функцию с полученными аргументами;
- выведет результат;
- вернет модифицированную функцию, готовую для использования.

Код будет выглядеть так:

```
>>> def document_it(func):
...     def new_function(*args, **kwargs):
...         print('Running function:', func.__name__)
...         print('Positional arguments:', args)
...         print('Keyword arguments:', kwargs)
```

```
...         result = func(*args, **kwargs)
...         print('Result:', result)
...         return result
...     return new_function
```

Независимо от того, какую функцию `func` вы передадите `document_it()`, вы получите новую функцию, которая содержит дополнительные выражения, добавляемые `document_it()`. Декоратор не обязательно должен запускать код функции `func`, но функция `document_it()` вызовет часть `func`, поэтому вы получите результат работы функции `func`, а также дополнительные данные:

```
>>> def add_ints(a, b):
...    return a + b
...
>>> add_ints(3, 5)
8
>>> cooler_add_ints = document_it(add_ints)  # мануальное присваивание декоратора
>>> cooler_add_ints(3, 5)
Running function: add_ints
Positional arguments: (3, 5)
Keyword arguments: {}
Result: 8
8
```

В качестве альтернативы мануальному присваиванию декоратора, показанному выше, просто добавьте конструкцию `@имя_декоратора` перед функцией, которую хотите декорировать:

```
>>> @document_it
... def add_ints(a, b):
...     return a + b
...
>>> add_ints(3, 5)
Start function add_ints
Positional arguments: (3, 5)
Keyword arguments: {}
Result: 8
8
```

Каждая функция может иметь более одного декоратора. Напишем еще один декоратор, который называется `square_it()` и возводит результат в квадрат.

```
>>> def square_it(func):
...     def new_function(*args, **kwargs):
...         result = func(*args, **kwargs)
...         return result * result
...     return new_function
...
```

Декоратор, размещенный ближе всего к функции (прямо над def), будет выполнен первым, а затем — тот, что находится сразу над ним. Любой порядок вызова вернет один и тот же конечный результат, но вы можете увидеть, как меняются промежуточные шаги:

```
>>> @document_it
... @square_it
... def add_ints(a, b):
...     return a + b
...
>>> add_ints(3, 5)
Running function: new_function
Positional arguments: (3, 5)
Keyword arguments: {}
Result: 64
64
```

Попробуем поменять порядок декораторов:

```
>>> @square_it
... @document_it
... def add_ints(a, b):
...     return a + b
...
>>> add_ints(3, 5)
Running function: add_ints
Positional arguments: (3, 5)
Keyword arguments: {}
Result: 8
64
```

Пространства имен и область определения

Имя может ссылаться на несколько разных вещей в зависимости от того, где оно используется. Программы в Python могут иметь разные *пространства имен* — разделы, внутри которых определенное имя уникально и не связано с такими же именами в других пространствах имен.

Каждая функция определяет собственное пространство имен. Если вы определите переменную, которая называется x в основной программе, и другую переменную x в отдельной функции, они будут ссылаться на разные значения. Но эту стену можно пробить: если нужно, вы можете получить доступ к именам других пространств имен разными способами.

В основной программе определяется *глобальное* пространство имен, поэтому переменные, находящиеся в этом пространстве имен, являются *глобальными*.

Вы можете получить значение глобальной переменной внутри функции:

```
>>> animal = 'fruitbat'
>>> def print_global():
...     print('inside print_global:', animal)
...
>>> print('at the top level:', animal)
at the top level: fruitbat
>>> print_global()
inside print_global: fruitbat
```

Но если попробуете получить значение глобальной переменной **и** изменить его внутри функции, получите ошибку:

```
>>> def change_and_print_global():
...     print('inside change_and_print_global:', animal)
...     animal = 'wombat'
...     print('after the change:', animal)
...
>>> change_and_print_global()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in change_and_report_it
UnboundLocalError: local variable 'animal' referenced before assignment
```

Если вы просто измените его, изменится другая переменная, которая также называется `animal`, но находится внутри функции:

```
>>> def change_local():
...     animal = 'wombat'
...     print('inside change_local:', animal, id(animal))
...
>>> change_local()
inside change_local: wombat 4330406160
>>> animal
'fruitbat'
>>> id(animal)
4330390832
```

Что здесь произошло? В первой строке мы присвоили строку `'fruitbat'` глобальной переменной с именем `animal`. Функция `change_local()` также имеет переменную с именем `animal`, но она находится в ее локальном пространстве имен.

Мы использовали функцию `id()`, чтобы вывести на экран уникальное значение каждого объекта и доказать, что переменная `animal`, расположенная внутри функции `change_local()`, — это не переменная `animal`, расположенная на основном уровне программы.

Чтобы получить доступ к глобальной переменной вместо локальной переменной внутри функции, вам нужно явно использовать ключевое слово `global` (вы знали, что я это скажу: явное лучше неявного):

```
>>> animal = 'fruitbat'
>>> def change_and_print_global():
...     global animal
...     animal = 'wombat'
...     print('inside change_and_print_global:', animal)
...
>>> animal
'fruitbat'
>>> change_and_print_global()
inside change_and_print_global: wombat
>>> animal
'wombat'
```

Если вы не используете ключевое слово `global` внутри функции, Python задействует локальное пространство имен и переменная будет локальной. Она пропадет после того, как функция завершит работу.

Python предоставляет две функции для доступа к содержимому ваших пространств имен:

- `locals()` — возвращает словарь, содержащий имена локального пространства имен;
- `globals()` — возвращает словарь, содержащий имена глобального пространства имен.

Вот так они используются:

```
>>> animal = 'fruitbat'
>>> def change_local():
...     animal = 'wombat'  # локальная переменная
...     print('locals:', locals())
...
>>> animal
'fruitbat'
>>> change_local()
locals: {'animal': 'wombat'}
>>> print('globals:', globals()) # немного переформатировано для представления
globals: {'animal': 'fruitbat',
'__doc__': None,
'change_local': <function change_it at 0x1006c0170>,
'__package__': None,
'__name__': '__main__',
'__loader__': <class '_frozen_importlib.BuiltinImporter'>,
'__builtins__': <module 'builtins'>}
>>> animal
'fruitbat'
```

Локальное пространство имен внутри функции change_local() содержало только локальную переменную animal. Глобальное пространство имен содержало отдельную глобальную переменную animal и многое другое.

Использование _ и __ в именах. Имена, которые начинаются с двух нижних подчеркиваний (__), зарезервированы для использования внутри Python, поэтому вам не следует применять их для своих переменных. Этот шаблон именования был выбран потому, что разработчики, скорее всего, не будут использовать его для создания имен своих переменных.

Например, имя функции находится в системной переменной *функция* .__name__, а имя ее строки документации — *функция* .__doc__:

```
>>> def amazing():
...     '''This is the amazing function.
...     Want to see it again?'''
...     print('This function is named:', amazing.__name__)
...     print('And its docstring is:', amazing.__doc__)
...
>>> amazing()
This function is named: amazing
And its docstring is: This is the amazing function.
    Want to see it again?
```

Как вы видели ранее в содержимом globals, основной программе присвоено специальное имя __main__.

Обработка ошибок с помощью try и except

Делай или не делай. Не надо пытаться.

Йода

В некоторых языках программирования ошибки отображаются с помощью специальных возвращаемых значений. В Python используются *исключения*: код, который выполняется, когда происходит связанная с ним ошибка.

Нечто похожее вы уже видели, когда попытались получить доступ к не входящей в список/кортеж позиции или ключу, которого не существует в словаре. Когда вы выполняете код, который при некоторых обстоятельствах может не сработать, вам также понадобятся *обработчики исключений*, чтобы перехватить любые потенциальные ошибки.

Хорошим тоном является использование обработчиков исключений везде, где может быть сгенерировано исключение, чтобы пользователь знал, что происходит. Вы можете быть неспособны исправить ошибку, но по крайней мере можете узнать,

при каких обстоятельствах это произошло, и аккуратно завершить программу. Если исключение сгенерировалось в функции и не было обработано, оно **всплывает** до тех пор, пока не будет поймано соответствующим обработчиком в одной из вызывающих функций. Если вы не предоставите собственный обработчик исключения, Python выведет сообщение об ошибке и некоторую информацию о том, где произошла ошибка, а затем завершит программу, как показано в следующем фрагменте кода.

```
>>> short_list = [1, 2, 3]
>>> position = 5
>>> short_list[position]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

Вместо того чтобы отпускать события на волю случая, размещайте свой код в блоке `try` и используйте блок `except`, чтобы обработать ошибку:

```
>>> short_list = [1, 2, 3]
>>> position = 5
>>> try:
...     short_list[position]
... except:
...     print('Need a position between 0 and', len(short_list)-1, ' but got',
...            position)
...
Need a position between 0 and 2 but got 5
```

Запускается код внутри блока `try`. Если произошла ошибка, генерируется исключение и выполняется код, расположенный внутри блока `except`. Если ошибок не произошло, блок `except` будет опущен.

Отсутствие аргументов в блоке `except`, как показано в предыдущем примере, позволяет ловить исключения любого типа. Если может сгенерироваться более одного исключения, лучшим решением будет предоставить отдельный обработчик для каждого из них. Никто не заставляет вас делать это — можете использовать блок `except` без аргументов, но ваша обработка будет более общей (что-то вроде вывода на экран строки Произошла ошибка). Вы можете использовать любое количество обработчиков исключений.

Иногда вам может понадобиться получить не только тип исключения. Вы можете получить объект исключения целиком в переменной `имя`, если используете следующую форму:

```
except тип_исключения as имя
```

В следующем примере выполняется проверка на `IndexError`, поскольку именно это исключение вызывается, когда вы предоставляете недействительную позицию последовательности. Исключение `IndexError` сохраняется в переменной `err`, а любое

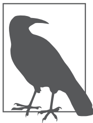
другое исключение — в переменной other. В примере на экран выводится все, что хранится в переменной other, чтобы показать, что вы получаете в этом объекте:

```
>>> short_list = [1, 2, 3]
>>> while True:
...     value = input('Position [q to quit]? ')
...     if value == 'q':
...         break
...     try:
...         position = int(value)
...         print(short_list[position])
...     except IndexError as err:
...         print('Bad index:', position)
...     except Exception as other:
...         print('Something else broke:', other)
...
Position [q to quit]? 1
2
Position [q to quit]? 0
1
Position [q to quit]? 2
3
Position [q to quit]? 3
Bad index: 3
Position [q to quit]? 2
3
Position [q to quit]? two
Something else broke: invalid literal for int() with base 10: 'two'
Position [q to quit]? q
```

Ввод позиции 3, как и ожидалось, генерирует исключение IndexError. Ввод слова two не понравился функции int(), которую мы обработали во втором, всеохватывающем обработчике.

Создание собственных исключений

В предыдущем разделе мы обсудили обработку исключений, но все исключения (такие как IndexError) заранее определены в Python или его стандартных библиотеках. Вы можете использовать любые из этих исключений в собственных интересах. Можете также определить собственные типы исключений, чтобы обрабатывать особые ситуации, которые могут возникнуть в ваших программах.



Для этого потребуется определить новый тип объекта с помощью класса — этим мы не будем заниматься вплоть до главы 6. Поэтому, если вы не знакомы с классами, может понадобиться вернуться к этому разделу позже.

Любое исключение является классом, в частности потомком класса `Exception`. Создадим исключение, которое называется `UppercaseException`, и вызовем его, когда встретим слово, записанное в верхнем регистре.

```
>>> class UppercaseException(Exception):
...     pass
...
>>> words = ['eeenie', 'meenie', 'miny', 'MO']
>>> for word in words:
...     if word.isupper():
...         raise UppercaseException(word)
...
Traceback (most recent call last):
  File "<stdin>", line 3, in <module>
__main__.UppercaseException: MO
```

Мы даже не определяли поведение исключения `UppercaseException` (обратите внимание на то, что мы просто использовали `pass`), позволив его родительскому классу `Exception` самостоятельно разобраться, что вывести на экран при генерации исключения.

Вы можете получить доступ к самому объекту исключения и вывести его на экран:

```
>>> try:
...     raise OopsException('panic')
... except OopsException as exc:
...     print(exc)
...
panic
```

Упражнения

1. Присвойте значение `7` переменной `guess_me`. Далее напишите условные проверки (`if`, `else` и `elif`), чтобы вывести строку `'too low'`, если значение переменной `guess_me` меньше `7`, `'too high'`, если оно больше `7`, и `'just right'`, если равно `7`.
2. Присвойте значение `7` переменной `guess_me` и значение `1` переменной `start`. Напишите цикл `while`, который сравнивает переменные `start` и `guess_me`. Выведите строку `'too low'`, если значение переменной `start` меньше значения переменной `guess_me`. Если значение переменной `start` равно значению переменной `guess_me`, выведите строку `'found it!'` и выйдите из цикла. Если значение переменной `start` больше значения переменной `guess_me`, выведите строку `'oops'` и выйдите из цикла. Увеличьте значение переменной `start` на выходе из цикла.
3. Используйте цикл `for`, чтобы вывести на экран значения списка `[3, 2, 1, 0]`.
4. Используйте включение списка, чтобы создать список, который содержит нечетные числа в диапазоне `range(10)`.

5. Используйте включение словаря, чтобы создать словарь squares. Используйте вызов range(10), чтобы получить ключи, и возведите их в квадрат, чтобы получить их значения.
6. Используйте включение множества, чтобы создать множество odd, которое содержит четные числа в диапазоне range(10).
7. Используйте включение генератора, чтобы вернуть строку 'Got' и количество чисел в диапазоне range(10). Итерируйте по нему с помощью цикла for.
8. Определите функцию good, которая возвращает список ['Harry', 'Ron', 'Hermione'].
9. Определите функцию генератора get_odds, которая возвращает четные числа из диапазона range(10). Используйте цикл for, чтобы найти и вывести третье возвращенное значение.
10. Определите декоратор test, который выводит строку 'start', когда вызывается функция, и строку 'end', когда функция завершает свою работу.
11. Определите исключение, которое называется OopsException. Сгенерируйте его, чтобы увидеть, что произойдет. Затем напишите код, позволяющий поймать это исключение и вывести строку 'Caught an oops'.
12. Используйте функцию zip(), чтобы создать словарь movies, который объединяет в пары эти списки: titles = ['Creature of Habit', 'Crewel Fate'] и plots = ['A nun turns into a monster', 'A haunted yarn shop'].

5 Py Boxes: модули, пакеты и программы

Вы уже прошли путь от встроенных типов данных до создания более крупных структур данных и кода. В этой главе вы наконец дойдете до самого главного и научитесь писать реалистичные и объемные программы на Python.

Отдельные программы

До этого момента вы писали и запускали с помощью интерактивного интерпретатора Python фрагменты кода вроде следующего:

```
>>> print("This interactive snippet works.")
This interactive snippet works.
```

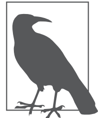
Теперь создадим вашу первую отдельную программу. Создайте файл под названием `test1.py`, содержащий следующую строку кода:

```
print("This standalone program works!")
```

Обратите внимание на отсутствие символов >>>, перед вами лишь одна строка кода. Убедитесь, что перед `print` нет пробелов.

Если вы работаете с Python с помощью текстовой консоли или окна терминала, введите имя вашей программы Python, а затем — имя файла:

```
$ python test1.py
This standalone program works!
```



Вы можете сохранить все фрагменты кода, которые встречаются в книге, в файлы и запустить их непосредственно. Если вы копируете их вместо того, чтобы набирать вручную, убедитесь, что удалили все символы >>> и ..., а также завершающий символ пробела.

Аргументы командной строки

Создайте файл test2.py, который содержит две следующие строки:

```
import sys
print('Program arguments:', sys.argv)
```

Теперь используйте свою версию Python, чтобы запустить эту программу. Вот так может выглядеть окно терминала в операционных системах Linux или Mac OS X, использующее стандартную программу оболочки:

```
$ python test2.py
Program arguments: ['test2.py']
$ python test2.py tra la la
Program arguments: ['test2.py', 'tra', 'la', 'la']
```

Модули и оператор import

Мы собираемся перейти на новый уровень — создание и использование кода более чем из одного файла. *Модуль* — это всего лишь файл, содержащий код Python.

Текст этой книги организован в иерархию: слова, предложения, абзацы и главы. В противном случае он стал бы нечитаемым спустя пару страниц. У кода имеется подобная организация: типы данных похожи на слова, операторы и выражения — это предложения, функции — это абзацы, а модули — это главы. Продолжу аналогию: когда я говорю, что что-то будет более подробно рассмотрено в главе 8, в программировании это было бы похоже на отсылку к коду другого модуля.

Мы ссылаемся на код других модулей с помощью оператора import. Оно позволяет получить доступ к коду и переменным этого модуля из вашей программы.

Импортируем модуль

Простейший вариант использования оператора import выглядит как import *модуль*, где *модуль* — это имя другого файла Python без расширения .py. Симулируем работу метеостанции и выведем на экран отчет о погоде. Основная программа выведет на экран отчет, а отдельный модуль, содержащий одну функцию, вернет описание погоды, которое будет использовано в отчете.

Основная программа выглядит так (назовем ее weatherman.py):

```
import report
description = report.get_description()
print("Today's weather:", description)
```

А ее модуль (report.py) — так:

```
def get_description():  # смотрите строку документации
    """Return random weather, just like the pros"""
```

```
    from random import choice
    possibilities = ['rain', 'snow', 'sleet', 'fog', 'sun', 'who knows']
    return choice(possibilities)
```

Если вы поместите оба этих файла в один каталог и укажете Python запустить файл `weatherman.py` в качестве основной программы, он обратится к модулю `report` и запустит его функцию `get_description()`. Мы написали эту версию функции `get_description()` так, чтобы она возвращала случайную строку из списка, которую выведет на экран основная программа:

```
$ python weatherman.py
Today's weather: who knows
$ python weatherman.py
Today's weather: sun
$ python weatherman.py
Today's weather: sleet
```

Мы использовали оператор `import` в двух местах.

- В основной программе `weatherman.py`, импортируемой модулем `report`.
- В файле модуля `report.py` функция `get_description()` импортирует функцию `choice` из стандартного модуля Python `random`.

Мы также использовали эти операторы двумя разными способами.

- Основная программа делала вызов `import report` и затем вызывала функцию `report.get_description()`.
- Функция `get_description()` из модуля `report.py` содержит вызовы `from random import choice` и `choice(possibilities)`.

В первом случае мы импортировали модуль `report` целиком, при этом нам нужно было добавить префикс `report.`, чтобы вызвать функцию `get_description()`. После этого оператора `import` все содержимое файла `report.py` становится доступным основной программе, нужно лишь ставить перед именем вызываемой функции префикс `report.`. Путем уточнения содержимого модуля с помощью его имени мы избегаем возникновения неприятных конфликтов именования. В каком-то другом модуле также может быть функция `get_descpirtion()`, и мы не вызовем ее по ошибке.

Во втором случае мы находимся внутри функции и знаем, что существует только одна функция с именем `choice`, поэтому импортируем функцию `choice()` непосредственно из модуля `random`. Мы могли бы написать функцию как следующий сниппет, который возвращает случайный результат:

```
def get_description():
    import random
    possibilities = ['rain', 'snow', 'sleet', 'fog', 'sun', 'who knows']
    return random.choice(possibilities)
```

Как и для многих других аспектов программирования, выбирайте стиль, который кажется вам наиболее прозрачным. Имя функции, перед которым стоит имя моду-

ля (random.choice), использовать безопаснее, однако из-за этого придется набирать немного больше текста.

Эти примеры применения функции get_description() продемонстрировали варианты того, **что** можно импортировать, но не показали, **где** следует выполнять импортирование, — в них import вызывался изнутри функции. Мы могли бы импортировать random из другой функции:

```
>>> import random
>>> def get_description():
...     possibilities = ['rain', 'snow', 'sleet', 'fog', 'sun', 'who knows']
...     return random.choice(possibilities)
...
>>> get_description()
'who knows'
>>> get_description()
'rain'
```

Вам следует рассмотреть возможность импортировать код вне функции, если импортируемый код может быть использован более одного раза, и изнутри функции, если вы знаете, что использование кода будет ограничено. Некоторые люди предпочитают размещать все операторы import в верхней части файла, чтобы явно обозначить все зависимости их кода. Оба варианта работают.

Импортируем модуль с другим именем

В нашей основной программе weatherman.py мы делали вызов import report. Но что, если у вас есть другой модуль с таким же именем или вы хотите использовать более короткое или простое имя? В такой ситуации можете выполнить импорт с помощью *псевдонима*. Используем псевдоним wr:

```
import report as wr
description = wr.get_description()
print("Today's weather:", description)
```

Импортируем только самое необходимое

С помощью Python вы можете импортировать одну или несколько частей модуля. Каждая часть может сохранить свое оригинальное имя, или же вы можете дать ей alias. Для начала импортируем функцию get_description() из модуля report с помощью его оригинального имени:

```
from report import get_description
description = get_description()
print("Today's weather:", description)
```

Теперь импортируем ее как do_it:

```
from report import get_description as do_it
description = do_it()
print("Today's weather:", description)
```

Каталоги поиска модулей

Где Python ищет файлы для импорта? Он использует список имен каталогов и ZIP-архив, хранящийся в стандартном модуле sys, как переменную path. Вы можете получить доступ к этому списку и изменить его. Вот так выглядит значение переменной sys.path в Python 3.3 в моей версии операционной системы Mac:

```
>>> import sys
>>> for place in sys.path:
...     print(place)
...
/Library/Frameworks/Python.framework/Versions/3.3/lib/python33.zip
/Library/Frameworks/Python.framework/Versions/3.3/lib/python3.3
/Library/Frameworks/Python.framework/Versions/3.3/lib/python3.3/plat-darwin
/Library/Frameworks/Python.framework/Versions/3.3/lib/python3.3/lib-dynload
/Library/Frameworks/Python.framework/Versions/3.3/lib/python3.3/site-packages
```

Пустое место в начале вывода содержит в себе строку '', которая символизирует текущий каталог. Если строка '' находится первой в sys.path, Python сначала выполнит поиск в текущем каталоге, когда вы попробуете что-то импортировать: import report выглядит как report.py.

Будет использован первый найденный модуль. Это означает, что, если вы определите модуль с именем random и он будет найден раньше оригинального модуля, вы не получите доступ к стандартной библиотеке random.

Пакеты

Мы перешли от отдельных строк кода к функциям, отдельным программам и модулям, располагающимся в одной папке. Чтобы сделать еще один шаг, вы можете организовать модули в иерархии файлов, которые называются *пакетами*.

Возможно, нам нужны разные типы прогнозов погоды: на следующий день и следующую неделю. В качестве одного из вариантов мы можем создать папку с именем sources, а внутри нее — два модуля: daily.py и weekly.py. Каждый из них содержит функцию forecast. Версия на каждый день возвращает строку, а версия на каждую неделю — список из семи строк.

Рассмотрим основную программу и два модуля. (Функция enumerate() разбивает список на части и отправляет каждый элемент списка в цикл for, добавляя к каждому элементу число в качестве небольшого бонуса.)

Основная программа — boxes/weather.py:

```
from sources import daily, weekly
print("Daily forecast:", daily.forecast())
print("Weekly forecast:")
for number, outlook in enumerate(weekly.forecast(), 1):
    print(number, outlook)
Модуль 1: boxes/sources/daily.py.
def forecast():
    'fake daily forecast'
    return 'like yesterday'
Модуль 2: boxes/sources/weekly.py.
def forecast():
    """Fake weekly forecast"""
    return ['snow', 'more snow', 'sleet',
        'freezing rain', 'rain', 'fog', 'hail']
```

В папке sources вам понадобится иметь кое-что еще — файл с именем __init__.py. Он может быть пустым, но Python он нужен для того, чтобы считать папку, которая его содержит, пакетом.

Запустите основную программу weather.py, чтобы увидеть, что произойдет:

```
$ python weather.py
Daily forecast: like yesterday
Weekly forecast:
1 snow
2 more snow
3 sleet
4 freezing rain
5 rain
6 fog
7 hail
```

Стандартная библиотека Python

Одно из основных преимуществ Python заключается в том, что у него есть собственный «запас мощности» — большая стандартная библиотека модулей, которые выполняют множество полезных задач и располагаются отдельно друг от друга, чтобы избежать разрастания ядра языка. Когда вы собираетесь писать код, зачастую сначала стоит проверить, существует ли стандартный модуль, который уже делает то, что вы хотите. Удивительно, как часто вы будете встречать эти небольшие жемчужины в стандартной библиотеке. Python также предоставляет авторитетную документацию для модулей наряду с руководством для пользователей (http://docs.python.org/3/library). Сайт Дага Хеллмана (Doug Hellmann) *Python Module of the Week* (http://bit.ly/py-motw) и его книга *The Python Standard*

Library by Example («Стандартная библиотека Python в примерах»), выпущенная издательством Addison-Wesley Professional, также являются очень полезными руководствами.

В следующих главах книги показано множество стандартных модулей, которые предназначены для работы с Сетью, системами, базами данных и т. д. В этом разделе я поговорю о стандартных модулях, которые имеют более общие варианты использования.

Обработка отсутствующих ключей с помощью функций setdefault() и defaultdict()

Вы уже видели, что попытка получить доступ к словарю с помощью несуществующего ключа генерирует исключение. Использование функции словаря `get()` для того, чтобы вернуть значение по умолчанию, помогает этого избежать.

Функция `setdefault()` похожа на функцию `get()`, но она также присваивает элемент словарю, если заданный ключ отсутствует:

```
>>> periodic_table = {'Hydrogen': 1, 'Helium': 2}
>>> print(periodic_table)
{'Helium': 2, 'Hydrogen': 1}
```

Если ключа еще нет в словаре, будет использовано новое значение:

```
>>> carbon = periodic_table.setdefault('Carbon', 12)
>>> carbon
12
>>> periodic_table
{'Helium': 2, 'Carbon': 12, 'Hydrogen': 1}
```

Если мы пытаемся присвоить другое значение по умолчанию уже существующему ключу, будет возвращено оригинальное значение и ничто не изменится:

```
>>> helium = periodic_table.setdefault('Helium', 947)
>>> helium
2
>>> periodic_table
{'Helium': 2, 'Carbon': 12, 'Hydrogen': 1}
```

Функция `defaultdict()` похожа на предыдущую, но она определяет значение по умолчанию для новых ключей заранее, при создании словаря. В этом примере мы передаем функцию `int`, которая будет вызываться как `int()`, и возвращаем значение `0`:

```
>>> from collections import defaultdict
>>> periodic_table = defaultdict(int)
```

Теперь любое отсутствующее значение будет заменяться целым числом (int) 0:

```
>>> periodic_table['Hydrogen'] = 1
>>> periodic_table['Lead']
0
>>> periodic_table
defaultdict(<class 'int'>, {'Lead': 0, 'Hydrogen': 1})
```

Аргументом defaultdict() является функция, возвращающая значение, которое будет присвоено отсутствующему ключу. В следующем примере функция no_idea() будет вызываться всякий раз, когда нужно вернуть значение:

```
>>> from collections import defaultdict
>>>
>>> def no_idea():
...     return 'Huh?'
...
>>> bestiary = defaultdict(no_idea)
>>> bestiary['A'] = 'Abominable Snowman'
>>> bestiary['B'] = 'Basilisk'
>>> bestiary['A']
'Abominable Snowman'
>>> bestiary['B']
'Basilisk'
>>> bestiary['C']
'Huh?'
```

Вы можете использовать функции int(), list() или dict(), чтобы возвращать пустые значения по умолчанию: int() возвращает 0, list() возвращает пустой список ([]) и dict() возвращает пустой словарь ({}). Если вы опустите аргумент, исходное значение нового ключа будет равно None.

Кстати, вы можете использовать lambda для того, чтобы определить функцию по умолчанию изнутри вызова:

```
>>> bestiary = defaultdict(lambda: 'Huh?')
>>> bestiary['E']
'Huh?'
```

Применение int — это один из способов создать ваш собственный прилавок:

```
>>> from collections import defaultdict
>>> food_counter = defaultdict(int)
>>> for food in ['spam', 'spam', 'eggs', 'spam']:
...     food_counter[food] += 1
...
>>> for food, count in food_counter.items():
...     print(food, count)
```

```
...
eggs 1
spam 3
```

В предыдущем примере, если бы food_counter был обычным словарем, а не defaultdict, Python генерировал бы исключение всякий раз, когда бы мы пытались увеличить элемент словаря food_counter[food], поскольку он был бы не инициализирован. Нам понадобилось бы сделать дополнительную работу, как показано здесь:

```
>>> dict_counter = {}
>>> for food in ['spam', 'spam', 'eggs', 'spam']:
...     if not food in dict_counter:
...         dict_counter[food] = 0
...     dict_counter[food] += 1
...
>>> for food, count in dict_counter.items():
...     print(food, count)
...
spam 3
eggs 1
```

Подсчитываем элементы с помощью функции Counter()

Если говорить о счетчиках, то в стандартной библиотеке имеется счетчик, который решает задачу, показанную в предыдущем примере, и даже больше:

```
>>> from collections import Counter
>>> breakfast = ['spam', 'spam', 'eggs', 'spam']
>>> breakfast_counter = Counter(breakfast)
>>> breakfast_counter
Counter({'spam': 3, 'eggs': 1})
```

Функция most_common() возвращает все элементы в убывающем порядке или лишь те элементы, количество которых больше, чем заданный аргумент count:

```
>>> breakfast_counter.most_common()
[('spam', 3), ('eggs', 1)]
>>> breakfast_counter.most_common(1)
[('spam', 3)]
```

Счетчики можно объединять. Для начала снова взглянем на содержимое breakfast_counter:

```
>>> breakfast_counter
>>> Counter({'spam': 3, 'eggs': 1})
```

Теперь мы создадим новый список, который называется lunch, и счетчик, который называется lunch_counter:

```
>>> lunch = ['eggs', 'eggs', 'bacon']
>>> lunch_counter = Counter(lunch)
>>> lunch_counter
Counter({'eggs': 2, 'bacon': 1})
```

Счетчики можно объединить с помощью оператора +:

```
>>> breakfast_counter + lunch_counter
Counter({'spam': 3, 'eggs': 3, 'bacon': 1})
```

Как вы можете догадаться, счетчики можно вычитать друг из друга с помощью оператора -. Что мы будем есть на завтрак, но не на обед?

```
>>> breakfast_counter – lunch_counter
Counter({'spam': 3})
```

О'кей, теперь узнаем, что мы можем съесть на обед, но не можем на завтрак:

```
>>> lunch_counter – breakfast_counter
Counter({'bacon': 1, 'eggs': 1})
```

По аналогии с множествами, показанными в главе 4, вы можете получить общие элементы с помощью оператора пересечения &:

```
>>> breakfast_counter & lunch_counter
Counter({'eggs': 1})
```

В результате пересечения был получен общий элемент ('eggs') с низким значением счетчика. Это имеет смысл: на завтрак у нас было только одно яйцо, поэтому указанное количество является общим.

Наконец, вы можете получить все элементы с помощью оператора объединения |:

```
>>> breakfast_counter | lunch_counter
Counter({'spam': 3, 'eggs': 2, 'bacon': 1})
```

Элемент 'eggs' снова оказался общим для обоих счетчиков. В отличие от сложения объединение не складывает счетчики, а выбирает тот, который имеет наибольшее значение.

Упорядочиваем по ключу с помощью OrderedDict()

Многие примеры кода, показанные в первых главах этой книги, демонстрируют, что порядок ключей в словаре нельзя предсказать: вы можете добавить в определенном

порядке ключи a, b и c, но функция keys() вернет результат "c, a, b". Рассмотрим модифицированный пример из главы 1:

```
>>> quotes = {
...     'Moe': 'A wise guy, huh?',
...     'Larry': 'Ow!',
...     'Curly': 'Nyuk nyuk!',
...     }
>>> for stooge in quotes:
...  print(stooge)
...
Larry
Curly
Moe
```

Словарь OrderedDict() запоминает порядок, в котором добавлялись ключи, и возвращает их в том же порядке с помощью итератора. Попробуем создать OrderedDict из последовательности кортежей вида «ключ — значение»:

```
>>> from collections import OrderedDict
>>> quotes = OrderedDict([
...     ('Moe', 'A wise guy, huh?'),
...     ('Larry', 'Ow!'),
...     ('Curly', 'Nyuk nyuk!'),
...     ])
>>>
>>> for stooge in quotes:
...     print(stooge)
...
Moe
Larry
Curly
```

Стек + очередь == deque

deque (произносится как «дэк») — это двухсторонняя очередь, которая имеет возможности стека и очереди. Она полезна, когда вы хотите добавить и удалить элементы с любого конца последовательности. В следующем примере мы будем двигаться с обоих концов слова к его середине, чтобы увидеть, является ли оно палиндромом. Функция popleft() удаляет крайний слева элемент deque и возвращает его, функция pop() удаляет крайний справа элемент и возвращает его. Вместе они двигаются с концов слова к его середине. Работа будет продолжаться до тех пор, пока крайние символы совпадают и пока не будет достигнута середина:

```
>>> def palindrome(word):
...     from collections import deque
...     dq = deque(word)
...     while len(dq) > 1:
```

```
...         if dq.popleft() != dq.pop():
...             return False
...     return True
...
...
>>> palindrome('a')
True
>>> palindrome('racecar')
True
>>> palindrome('')
True
>>> palindrome('radar')
True
>>> palindrome('halibut')
False
```

Я воспользовался этим примером, чтобы было проще проиллюстрировать работу `deque`. Если вы действительно хотите создать программу, которая определяет палиндромы, гораздо проще было бы сравнивать строку с ее копией, вывернутой наизнанку. В Python строковой функции `reverse()` не существует, но можно обратить строку с помощью разбиения, как показано здесь:

```
>>> def another_palindrome(word):
...     return word == word[::-1]
...
>>> another_palindrome('radar')
True
>>> another_palindrome('halibut')
False
```

Итерируем по структурам кода с помощью itertools

`itertools` содержит особые функции итератора. Каждая из них возвращает один элемент при каждом вызове из цикла `for … in` и запоминает свое состояние между вызовами.

Функция `chain()` проходит по своим аргументам, как если бы они были единым итерабельным объектом:

```
>>> import itertools
>>> for item in itertools.chain([1, 2], ['a', 'b']):
...     print(item)
...
1
2
a
b
```

Функция `cycle()` является бесконечным итератором, проходящим в цикле по своим аргументам:

```
>>> import itertools
>>> for item in itertools.cycle([1, 2]):
...     print(item)
...
1
2
1
2
.
.
.
```

...и т. д.

Функция `accumulate()` подсчитывает накопленные значения. По умолчанию она высчитывает сумму:

```
>>> import itertools
>>> for item in itertools.accumulate([1, 2, 3, 4]):
...     print(item)
...
1
3
6
10
```

В качестве второго аргумента функции `accumulate()` вы можете передать функцию, и она будет использована вместо сложения. Функция должна принимать два аргумента и возвращать одно значение. В этом примере высчитывается произведение:

```
>>> import itertools
>>> def multiply(a, b):
...     return a * b
...
>>> for item in itertools.accumulate([1, 2, 3, 4], multiply):
...     print(item)
...
1
2
6
24
```

Модуль `itertools` имеет еще много функций, он известен благодаря определенным комбинациям и преобразованиям, которые могут сохранить кучу времени, если в них появится необходимость.

Выводим данные на экран красиво с помощью функции pprint()

Все наши примеры использовали функцию print() (или просто имя переменной в интерактивном интерпретаторе), чтобы выводить информацию на экран. Иногда результаты было трудно прочитать. Нам нужен *pretty printer* (красивый принтер) вроде pprint():

```
>>> from pprint import pprint
>>> quotes = OrderedDict([
...     ('Moe', 'A wise guy, huh?'),
...     ('Larry', 'Ow!'),
...     ('Curly', 'Nyuk nyuk!'),
...     ])
>>>
```

Старая добрая функция print() просто выводит всю информацию:

```
>>> print(quotes)
OrderedDict([('Moe', 'A wise guy, huh?'), ('Larry', 'Ow!'), ('Curly', 'Nyuk nyuk!')])
```

А функция pprint() пытается выровнять элементы для лучшей читаемости:

```
>>> pprint(quotes)
{'Moe': 'A wise guy, huh?',
 'Larry': 'Ow!',
 'Curly': 'Nyuk nyuk!'}
```

Нужно больше кода

Иногда в стандартной библиотеке нет нужной вам функциональности или же она реализована не так, как вам нужно. В этом случае можете воспользоваться целым миром стороннего программного обеспечения с открытым исходным кодом. Отлично зарекомендовали себя следующие ресурсы:

- PyPi (известный также как Cheese Shop («Сырный магазин»), он назван в честь старого Monty Python skit) (http://bit.ly/py-libex);
- github (https://github.com/Python);
- readthedocs (https://readthedocs.org/).

Небольшие фрагменты кода вы можете найти по адресу http://code.activestate.com/recipes/langs/python/.

Почти весь код Python использует функции стандартных библиотек Python. Кое-где показаны внешние пакеты: я упоминал requests в главе 1, а в подразделе «За пределами стандартной библиотеки: Requests» раздела «Веб-клиенты» главы 9 приведу более подробную информацию. В приложении Г показано, как устанавливать стороннее программное обеспечение Python, а также рассмотрены основные детали разработки.

Упражнения

1. Создайте файл, который называется `zoo.py`. В нем объявите функцию `hours()`, которая выводит на экран строку `' Open 9-5 daily'`. Далее используйте интерактивный интерпретатор, чтобы импортировать модуль zoo и вызвать его функцию `hours()`.
2. В интерактивном интерпретаторе импортируйте модуль zoo под именем menagerie и вызовите его функцию `hours()`.
3. Оставаясь в интерпретаторе, импортируйте непосредственно функцию `hours()` из модуля zoo и вызовите ее.
4. Импортируйте функцию `hours()` под именем `info` и вызовите ее.
5. Создайте словарь с именем `plain`, содержащий пары «ключ — значение» `'a': 1`, `'b': 2` и `'c':3`, а затем выведите его на экран.
6. Создайте `OrderedDict` с именем `fancy` из пар «ключ — значение», приведенных в упражнении 5, и выведите его на экран. Изменился ли порядок ключей?
7. Создайте `defaultdict` с именем `dict_of_lists` и передайте ему аргумент `list`. Создайте список `dict_of_lists['a']` и присоедините к нему значение `'something for a'` за одну операцию. Выведите на экран `dict_of_lists['a']`.

6 Ой-ой-ой: объекты и классы

Таинственных объектов не бывает. Они такими просто кажутся.

Элизабет Боуэн

Возьмите объект. Сделайте что-нибудь с ним. Добавьте что-нибудь другое к нему.

Джаспер Джонс

К этому моменту вы уже познакомились с такими структурами данных, как строки и словари, а также со структурами кода — функциями и модулями. В текущей главе вы узнаете о пользовательской структуре данных — *объектах*.

Что такое объекты

Как я упоминал в главе 2, все в Python, от чисел до модулей, является объектами. Однако Python скрывает бóльшую часть принципов функционирования объектов с помощью особого синтаксиса. Вы можете написать `num = 7`, чтобы создать объект типа `int` со значением `7`, и присвоить ссылку на него по имени `num`. Заглядывать внутрь объектов нужно только в случае, если вам необходимо создать собственный объект или модифицировать поведение уже существующих объектов. В этой главе вы увидите, как сделать и то и другое.

Объект содержит как данные (переменные, которые называются *атрибутами*), так и код (функции, которые называются *методами*). Он представляет собой уникальный экземпляр какого-то конкретного предмета. Например, целочисленный объект со значением `7` может использовать методы вроде сложения и умножения, что показано в разделе «Числа» главы 2. `8` — это другой объект. Это значит, что существует класс `Integer`, которому принадлежат объекты `7` и `8`. Строки `'cat'` и `'duck'` также являются объектами и имеют методы, с которыми вы уже знакомы, — например, `capitalize()` и `replace()`.

Когда вы создаете новые объекты, которые до вас не создавал никто, вы должны создать класс, который демонстрирует их содержимое.

Объекты можно считать существительными, а их методы — глаголами. Объект представляет отдельную вещь, а его методы определяют, как она взаимодействует с другими вещами.

В отличие от модулей, вы можете иметь одновременно несколько объектов, каждый из которых имеет разные значения атрибутов. Они являются суперструктурами данных, в которые был добавлен код.

Определяем класс с помощью ключевого слова class

В главе 1 я сравнил объект с пластмассовой коробкой. *Класс* похож на форму, из которой создается эта коробка. Например, `String` является встроенным классом Python, который создает строковые объекты вроде `'cat'` и `'duck'`. Python имеет множество других встроенных классов, позволяющих создавать другие стандартные типы данных, включая списки, словари и т. д. Чтобы создать собственный объект в Python, вам сначала нужно определить класс с помощью ключевого слова `class`. Рассмотрим простой пример.

Предположим, вы хотите определять объекты, которые представляют информацию о людях. Каждый объект будет представлять одного человека. Сначала вам нужно определить класс `Person` в качестве формы. В последующих примерах мы попробуем использовать больше версий этого класса по мере продвижения от простейшего класса к классу, который действительно может делать что-то полезное.

В первый раз создадим самый простой из возможных классов — пустой класс:

```
>>> class Person():
...     pass
```

Как и в случае с функциями, нам нужно сказать `pass`, чтобы показать, что этот класс пуст. Такое определение является необходимым минимумом создания объекта. Вы создаете объект из класса с помощью вызова имени класса так, будто оно является функцией:

```
>>> someone = Person()
```

В этом случае `Person()` создает отдельный объект класса `Person` и присваивает его имени `someone`. Но наш класс `Person` пуст, поэтому объект `someone`, который мы создали, просто занимает место и ничего не делает. Вы никогда не будете определять такой класс, я показываю его только для того, чтобы на его основе создать следующий пример.

Попробуем снова — в этот раз добавим в класс специальный метод инициализации `__init__`:

```
>>> class Person():
...     def __init__(self):
...         pass
```

Так выглядят реальные определения классов в Python. Согласен, `__init__()` и `self` смотрятся странно. `__init__()` — это особое имя метода, который инициализирует отдельный объект с помощью определения его класса[1]. Аргумент `self` указывает на сам объект.

Когда вы указываете `__init__()` в определении класса, его первым параметром должен быть объект `self`. Несмотря на то что в Python `self` не является зарезервированным словом, оно применяется довольно часто. Никому из тех, кто будет читать ваш код позже (включая вас!), не придется гадать, что вы имели в виду, когда использовали слово `self`.

Но даже в этом случае второе определение класса `Person` создает объект, который ничего не делает. Наша третья попытка покажет, насколько легко создать простой объект в Python. В этот раз мы добавим параметр `name` в метод инициализации:

```
>>> class Person():
...     def __init__(self, name):
...         self.name = name
...
>>>
```

Теперь мы можем создать объект класса `Person`, передав строку для параметра `name`:

```
>>> hunter = Person('Elmer Fudd')
```

Эта строка кода делает следующее:

- выполняет поиск определения класса `Person`;
- *инстанцирует* (создает) новый объект в памяти;
- вызывает метод объекта `__init__`, передавая только что созданный объект под именем `self` и другой аргумент (`'Elmer Fudd'`) в качестве значения параметра `name`;
- сохраняет в объекте значение переменной `name`;
- возвращает новый объект;
- прикрепляет к объекту имя `hunter`.

Этот новый объект похож на любой другой объект Python. Вы можете использовать его как элемент списка, кортежа, словаря или множества. Можете передать его в функцию как аргумент или вернуть его в качестве результата.

Что насчет значения `name`, которое мы передали? Оно было сохранено как атрибут объекта. Вы можете прочитать и записать его непосредственно:

```
>>> print('The mighty hunter: ', hunter.name)
The mighty hunter: Elmer Fudd
```

[1] Вы встретите множество примеров двойных подчеркиваний в именах Python.

Помните, внутри определения класса `Person` вы получаете доступ к атрибуту `name` с помощью конструкции `self.name`. Когда вы создаете реальный объект вроде `hunter`, то ссылаетесь на этот атрибут как `hunter.name`.

Не обязательно иметь метод `__init__` в описании каждого класса, он используется для того, чтобы различать объекты одного класса.

Наследование

Может случиться, что, пытаясь решить какую-то задачу, вы обнаружите, что уже существует класс, создающий объекты, которые делают почти все из того, что вам нужно. Что вы можете предпринять? Вы можете модифицировать этот старый класс, но при этом сделаете его сложнее и можете сломать что-то, что раньше работало.

Конечно, можно написать новый класс, скопировав и вставив содержимое старого класса, а затем дополнить его новым кодом. Но это значит, что вам придется поддерживать больше кода и части старого и нового классов могут быть непохожими друг на друга, поскольку теперь находятся в разных местах.

Решением проблемы является *наследование* — создание нового класса из уже существующего, который при этом содержит какие-то дополнения и изменения. Это отличный способ использовать код повторно. Когда вы применяете наследование, новый класс может автоматически использовать весь код старого класса и при этом вам не нужно его копировать.

Вы определяете только то, что вам нужно добавить или изменить в новом классе, и этот код переопределяет поведение старого класса. Оригинальный класс называется *предком*, *суперклассом* или *базовым классом*, новый класс называется *потомком*, *подклассом* или *классом-наследником*. Эти термины в объектно-ориентированном программировании взаимозаменяемы.

Давайте же что-нибудь унаследуем. Мы определим пустой класс, который называется `Car`. Далее определим подкласс класса `Car`, который называется `Yugo`. Вы определяете подкласс с помощью все того же ключевого слова `class`, но указывая внутри скобок имя родительского класса (`class Yugo(Car)`, как показано ниже):

```
>>> class Car():
...     pass
...
>>> class Yugo(Car):
...     pass
...
```

Далее создадим объекты каждого класса:

```
>>> give_me_a_car = Car()
>>> give_me_a_yugo = Yugo()
```

Класс-потомок является уточненной версией класса-предка; если говорить в терминах объектно-ориентированных языков, Yugo **является** Car. Объект с именем give_me_a_yugo является экземпляром класса Yugo, но он также наследует все то, что может делать класс Car. В нашем случае классы Car и Yugo полезны как мертвому припарки, поэтому попробуем указать их новые определения, которые действительно могут что-то сделать:

```
>>> class Car():
...     def exclaim(self):
...         print("I'm a Car!")
...
>>> class Yugo(Car):
...     pass
...
```

Наконец, создадим по одному объекту каждого класса и вызовем их методы exclaim:

```
>>> give_me_a_car = Car()
>>> give_me_a_yugo = Yugo()
>>> give_me_a_car.exclaim()
I'm a Car!
>>> give_me_a_yugo.exclaim()
I'm a Car!
```

Не сделав ничего особенного, класс Yugo унаследовал метод exclaim() класса Car. Фактически класс Yugo говорит, что он является классом Car, что может привести к кризису самоопределения. Посмотрим, что мы можем с этим сделать.

Перегрузка метода

Как вы только что увидели, новый класс наследует все, что находится в его классе-предке. Далее вы увидите, как можно заменить, или *перегрузить*, родительские методы. Класс Yugo должен как-то отличаться от класса Car, иначе зачем вообще создавать новый класс. Изменим способ работы метода exclaim() для класса Yugo:

```
>>> class Car():
...     def exclaim(self):
...         print("I'm a Car!")
...
>>> class Yugo(Car):
...     def exclaim(self):
...         print("I'm a Yugo! Much like a Car, but more Yugo-ish.")
...
```

Теперь создадим объекты этих классов:

```
>>> give_me_a_car = Car()
>>> give_me_a_yugo = Yugo()
```

Что они говорят?

```
>>> give_me_a_car.exclaim()
I'm a Car!
>>> give_me_a_yugo.exclaim()
I'm a Yugo! Much like a Car, but more Yugo-ish.
```

В этих примерах мы перегрузили метод `exclaim()`. Перегрузить можно любые методы, включая `__init__()`. Рассмотрим другой пример, который использует наш более старый класс `Person`. Создадим подклассы, которые представляют докторов (`MDPerson`) и адвокатов (`JDPerson`):

```
>>> class Person():
...     def __init__(self, name):
...         self.name = name
...
>>> class MDPerson(Person):
...     def __init__(self, name):
...         self.name = "Doctor " + name
...
>>> class JDPerson(Person):
...     def __init__(self, name):
...         self.name = name + ", Esquire"
...
```

В этих случаях метод инициализации `__init__()` принимает те же аргументы, что и родительский класс `Person`, но внутри объекта сохраняет значение переменной `name` разными способами:

```
>>> person = Person('Fudd')
>>> doctor = MDPerson('Fudd')
>>> lawyer = JDPerson('Fudd')
>>> print(person.name)
Fudd
>>> print(doctor.name)
Doctor Fudd
>>> print(lawyer.name)
Fudd, Esquire
```

Добавление метода

В класс-потомок можно также добавить метод, которого не было в родительском классе. Возвращаясь к классам `Car` и `Yugo`, мы определим новый метод `need_a_push()` только для класса `Yugo`:

```
>>> class Car():
...     def exclaim(self):
...         print("I'm a Car!")
```

```
...
>>> class Yugo(Car):
...     def exclaim(self):
...         print("I'm a Yugo! Much like a Car, but more Yugo-ish.")
...     def need_a_push(self):
...         print("A little help here?")
...
```

Далее создадим объекты классов Car и Yugo:

```
>>> give_me_a_car = Car()
>>> give_me_a_yugo = Yugo()
```

Объект класса Yugo может реагировать на вызов метода need_a_push():

```
>>> give_me_a_yugo.need_a_push()
A little help here?
```

А объект общего класса Car — нет:

```
>>> give_me_a_car.need_a_push()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Car' object has no attribute 'need_a_push'
```

К этому моменту класс Yugo может делать что-то, чего не может делать класс Car, и теперь мы точно можем определить отдельную личность класса Yugo.

Просим помощи у предка с помощью ключевого слова super

Мы видели, как класс-потомок может добавить или перегрузить метод класса-предка. Но что, если вам нужно вызвать оригинальный метод родительского класса? «Рад, что вы спросили», — говорит метод super(). Мы определим новый класс, который называется EmailPerson и представляет объект класса Person, содержащий адрес электронной почты. Для начала запишем наше привычное определение класса Person:

```
>>> class Person():
...     def __init__(self, name):
...         self.name = name
...
```

Обратите внимание на то, что вызов метода __init__() в следующем подклассе имеет дополнительный параметр email:

```
>>> class EmailPerson(Person):
...     def __init__(self, name, email):
...         super().__init__(name)
...         self.email = email
```

Когда вы определяете метод `__init__()` для своего класса, вы заменяете метод `__init__()` родительского класса, который больше не вызывается автоматически. В результате вам нужно вызывать его явно. Происходит следующее.

- Метод `super()` получает определение родительского класса `Person`.
- Метод `__init__()` вызывает метод `Person.__init__()`. Последний заботится о том, чтобы передать аргумент `self` суперклассу, поэтому вам нужно лишь передать опциональные аргументы. В нашем случае единственным аргументом класса `Person()` будет `name`.
- Строка `self.email = email` — это новый код, который отличает класс `EmailPerson` от класса `Person`.

Теперь создадим одну персону:

```
>>> bob = EmailPerson('Bob Frapples', 'bob@frapples.com')
```

Мы должны иметь доступ к атрибутам `name` и `email`:

```
>>> bob.name
'Bob Frapples'
>>> bob.email
'bob@frapples.com'
```

Почему бы нам просто не определить новый класс так, как показано далее?

```
>>> class EmailPerson(Person):
...     def __init__(self, name, email):
...         self.name = name
...         self.email = email
```

Мы могли бы сделать это, но в таком случае потеряли бы возможность применять наследование. Мы использовали метод `super()`, чтобы создать объект, который работает примерно так же, как и объект класса `Person`. Есть и другое преимущество: если определение класса `Person` в будущем изменится, с помощью метода `super()` мы сможем гарантировать, что атрибуты и методы, которые класс `EmailPerson` наследует от класса `Person`, отреагируют на изменения.

Используйте метод `super()`, когда потомок делает что-то самостоятельно, но ему все еще нужно что-то от предка (как и в реальной жизни).

В защиту self

Python критикуют за то, что, помимо применения пробелов, необходимо включать `self` в качестве первого аргумента методов экземпляра класса (методов, которые вы видели в предыдущем примере). Python использует аргумент `self`, чтобы найти атрибуты и методы правильного объекта. Например, я покажу, как вы можете вызвать метод объекта и что Python сделает при этом за кулисами.

Помните класс Car из предыдущих примеров? Снова вызовем метод exclaim():

```
>>> car = Car()
>>> car.exclaim()
I'm a Car!
```

Вот что происходит за кулисами Python.

- Выполняется поиск класса (Car) объекта car.
- Объект car передается методу exclaim() класса Car как параметр self.

Ради забавы вы и сами можете запустить пример таким образом, и он сработает точно так же, как и нормальный синтаксис (car.exclaim()):

```
>>> Car.exclaim(car)
I'm a Car!
```

Однако нет причин использовать такой более длинный стиль.

Получаем и устанавливаем значение атрибутов с помощью свойств

Отдельные объектно-ориентированные языки поддерживают закрытые атрибуты объектов, к которым нельзя получить доступ непосредственно; программистам зачастую приходится писать *геттеры* и *сеттеры*, чтобы считать и записать значения таких атрибутов.

В Python геттеры и сеттеры не нужны, поскольку все атрибуты и методы являются открытыми, а от вас ожидается примерное поведение. Если прямой доступ к атрибутам заставляет вас нервничать, вы, конечно, можете написать геттеры и сеттеры. Но сделайте это более характерным для Python способом — используйте *свойства*.

В этом примере мы определим класс Duck, имеющий один атрибут hidden_name. (В следующем разделе я покажу вам более удачный способ именовать атрибуты, которые вы хотите оставить закрытыми.) Мы не хотим, чтобы люди обращались к атрибуту напрямую, поэтому определим два метода: геттер (get_name()) и сеттер (set_name()). Я добавил выражение print() в каждый из них, чтобы показать момент его вызова. Наконец, мы определим эти методы как свойства атрибута name:

```
>>> class Duck():
...     def __init__(self, input_name):
...         self.hidden_name = input_name
...     def get_name(self):
...         print('inside the getter')
...         return self.hidden_name
...     def set_name(self, input_name):
...         print('inside the setter')
...         self.hidden_name = input_name
...     name = property(get_name, set_name)
```

Новые методы действуют как обычные геттеры и сеттеры до последней строки, где они указываются как свойства атрибута `name`. Первый аргумент функции `property()` — это геттер, а второй — это сеттер. Теперь, когда вы обращаетесь к атрибуту `name` любого объекта `Duck`, вызывается метод `get_name()`, который возвращает его:

```
>>> fowl = Duck('Howard')
>>> fowl.name
inside the getter
'Howard'
```

Вы все еще можете вызвать метод `get_name()` непосредственно, как обычный геттер:

```
>>> fowl.get_name()
inside the getter
'Howard'
```

Когда вы присваиваете значение атрибуту `name`, вызывается метод `set_name()`:

```
>>> fowl.name = 'Daffy'
inside the setter
>>> fowl.name
inside the getter
'Daffy'
```

Метод `set_name()` вы также можете вызвать непосредственно:

```
>>> fowl.set_name('Daffy')
inside the setter
>>> fowl.name
inside the getter
'Daffy'
```

Еще один способ определить свойства — это *декораторы*. В следующем примере мы определим два разных метода с именем `name()`, предшествовать которым будут разные декораторы:

- `@property`, который размещается перед геттером;
- `@name.setter`, который размещается перед сеттером.

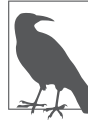
В коде они выглядят так:

```
>>> class Duck():
...     def __init__(self, input_name):
...         self.hidden_name = input_name
...     @property
...     def name(self):
...         print('inside the getter')
```

```
...         return self.hidden_name
...     @name.setter
...     def name(self, input_name):
...         print('inside the setter')
...         self.hidden_name = input_name
```

Вы все еще можете получать доступ к атрибуту `name`, но в этом случае не существует видимых методов `get_name()` или `set_name()`:

```
>>> fowl = Duck('Howard')
>>> fowl.name
inside the getter
'Howard'
>>> fowl.name = 'Donald'
inside the setter
>>> fowl.name
inside the getter
'Donald'
```



Если кто-то догадается, что мы называли наш атрибут hidden_name, он сможет считать и записать его непосредственно с помощью конструкции fowl.hidden_name. В следующем разделе вы увидите особый способ именования закрытых атрибутов в Python.

В обоих предыдущих примерах мы использовали свойство `name`, чтобы обратиться к отдельному атрибуту (в нашем случае `hidden_name`), который хранится внутри объекта. Свойство может ссылаться и на *вычисляемое значение*. Определим класс `Circle`, который имеет атрибут `radius` и вычисляемое свойство `diameter`:

```
>>> class Circle():
...     def __init__(self, radius):
...         self.radius = radius
...     @property
...     def diameter(self):
...         return 2 * self.radius
...
```

Мы создаем объект класса `Circle`, задав значение его атрибута `radius`:

```
>>> c = Circle(5)
>>> c.radius
5
```

Мы можем обратиться к свойству `diameter` точно так же, как к атрибуту вроде `radius`:

```
>>> c.diameter
10
```

А вот и самое интересное — мы можем изменить значение атрибута radius в любой момент и свойство diameter будет рассчитано на основе текущего значения атрибута radius:

```
>>> c.radius = 7
>>> c.diameter
14
```

Если вы не укажете сеттер для атрибута, то не сможете устанавливать его значение извне. Это удобно для атрибутов, которые должны быть доступны только для чтения:

```
>>> c.diameter = 20
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: can't set attribute
```

У использования свойств вместо непосредственного доступа к атрибутам имеется еще одно преимущество: если вы измените определение атрибута, вам нужно будет поправить только код внутри определения класса вместо того, чтобы править все вызовы.

Искажение имен для безопасности

В примере с классом Duck из предыдущего раздела мы вызывали наш (не полностью) скрытый атрибут hidden_name. Python предлагает соглашения по именованию для атрибутов, которые не должны быть видимы за пределами определения их классов: имена начинаются с двух нижних подчеркиваний (__).

Переименуем атрибут hidden_name в __name, как показано здесь:

```
>>> class Duck():
...     def __init__(self, input_name):
...         self.__name = input_name
…     @property
...     def name(self):
...         print('inside the getter')
...         return self.__name
...     @name.setter
...     def name(self, input_name):
...         print('inside the setter')
...         self.__name = input_name
...
```

Теперь проверим, работает ли все как полагается:

```
>>> fowl = Duck('Howard')
>>> fowl.name
inside the getter
```

```
'Howard'
>>> fowl.name = 'Donald'
inside the setter
>>> fowl.name
inside the getter
'Donald'
```

Выглядит хорошо. И вы не можете получить доступ к атрибуту `__name`:

```
>>> fowl.__name
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Duck' object has no attribute '__name'
```

Это соглашение по именованию не делает атрибут закрытым, но Python искажает имя для того, чтобы внешний код не наткнулся на него. Если вам любопытно и вы никому не расскажете, я покажу вам, как будет выглядеть атрибут:

```
>>> fowl._Duck__name
'Donald'
```

Обратите внимание на то, что на экране не появилась надпись inside the getter. Хотя эта защита не идеальна, искаженное имя отказывается случайно или намеренно получать доступ к атрибуту.

Типы методов

Одни данные *(атрибуты)* и функции *(методы)* являются частью самого класса, а другие — частью объектов, которые созданы на его основе.

Когда вы видите начальный аргумент `self` в методах внутри определения класса, этот метод является *методом экземпляра*. Такие методы вы обычно пишете при создании собственного класса. Первый параметр метода экземпляра — это `self`, и Python передает объект методу, когда вы его вызываете.

В противоположность ему *метод класса* влияет на весь класс целиком. Любое изменение, которое происходит с классом, влияет на все его объекты. Внутри определения класса декоратор `@classmethod` показывает, что следующая функция является методом класса. Первым параметром метода также является сам класс. Согласно традиции этот параметр называется `cls`, поскольку слово `class` является зарезервированным и не может быть использовано здесь. Определим метод класса для `A`, который будет подсчитывать количество созданных объектов:

```
>>> class A():
...     count = 0
...     def __init__(self):
...         A.count += 1
...     def exclaim(self):
...         print("I'm an A!")
```

```
...     @classmethod
...     def kids(cls):
...         print("A has", cls.count, "little objects.")
...
>>>
>>> easy_a = A()
>>> breezy_a = A()
>>> wheezy_a = A()
>>> A.kids()
A has 3 little objects.
```

Обратите внимание на то, что мы вызвали метод `A.count` (атрибут класса) вместо `self.count` (который является атрибутом объекта). В методе `kids()` мы использовали вызов `cls.count`, но с тем же успехом могли бы применять вызов `A.count`.

Третий тип методов не влияет ни на классы, ни на объекты: он находится внутри класса только для удобства вместо того, чтобы располагаться где-то отдельно. Это *статический метод*, перед которым располагается декоратор `@staticmethod`, не имеющий в качестве начального параметра ни `self`, ни класс `class`. Рассмотрим пример, который служит в качестве рекламы класса `CoyoteWeapon`:

```
>>> class CoyoteWeapon():
...     @staticmethod
...     def commercial():
...         print('This CoyoteWeapon has been brought to you by Acme')
...
>>>
>>> CoyoteWeapon.commercial()
This CoyoteWeapon has been brought to you by Acme
```

Обратите внимание на то, что нам не нужно создавать объект класса `CoyoteWeapon`, чтобы получить доступ к этому методу. Это здорово.

Утиная типизация

В Python имеется также реализация *полиморфизма* — это значит, что одна операция может быть произведена над разными объектами независимо от их класса.

Используем уже знакомый нам инициализатор `__init__()` для всех трех классов `Quote`, но добавим две новые функции:

- `who()` возвращает значение сохраненной строки `person`;
- `says()` возвращает сохраненную строку `words`, имеющую особую пунктуацию.

Посмотрим на них в действии:

```
>>> class Quote():
...     def __init__(self, person, words):
...         self.person = person
```

```
...         self.words = words
...     def who(self):
...         return self.person
...     def says(self):
...         return self.words + '.'
...
>>> class QuestionQuote(Quote):
...      def says(self):
...          return self.words + '?'
...
>>> class ExclamationQuote(Quote):
...      def says(self):
...          return self.words + '!'
...
>>>
```

Мы не меняли способ инициализации классов QuestionQuote и ExclamationQuote, поэтому не перегружали их методы __init__(). Далее Python автоматически вызывает метод __init__() родительского класса Quote, чтобы сохранить переменные объекта person и words. Поэтому мы можем получить доступ к атрибуту self.words в объектах, созданных с помощью подклассов QuestionQuote и ExclamationQuote.

Далее создадим несколько объектов:

```
>>> hunter = Quote('Elmer Fudd', "I'm hunting wabbits")
>>> print(hunter.who(), 'says:', hunter.says())
Elmer Fudd says: I'm hunting wabbits.
>>> hunted1 = QuestionQuote('Bugs Bunny', "What's up, doc")
>>> print(hunted1.who(), 'says:', hunted1.says())
Bugs Bunny says: What's up, doc?
>>> hunted2 = ExclamationQuote('Daffy Duck', "It's rabbit season")
>>> print(hunted2.who(), 'says:', hunted2.says())
Daffy Duck says: It's rabbit season!
```

Три разные версии метода says() обеспечивают разное поведение трех классов. Так выглядит традиционный полиморфизм в объектно-ориентированных языках. Python пошел немного дальше и позволяет вам вызывать методы who() и says() для **любых** объектов, включающих эти методы. Определим класс BabblingBrook, который не имеет никакого отношения к нашим охотнику и его жертвам (наследникам класса Quote), созданным ранее:

```
>>> class BabblingBrook():
...     def who(self):
...         return 'Brook'
...     def says(self):
...         return 'Babble'
...
>>> brook = BabblingBrook()
```

Теперь запустим методы who() и says() разных объектов, один из которых (brook) совершенно не связан с остальными:

```
>>> def who_says(obj):
...     print(obj.who(), 'says', obj.says())
...
>>> who_says(hunter)
Elmer Fudd says I'm hunting wabbits.
>>> who_says(hunted1)
Bugs Bunny says What's up, doc?
>>> who_says(hunted2)
Daffy Duck says It's rabbit season!
>>> who_says(brook)
Brook says Babble
```

Такое поведение иногда называется *утиной типизацией* благодаря старой поговорке «Если нечто выглядит как утка, плавает как утка и крякает как утка, то это, вероятно, утка и есть».

Особые методы

Теперь вы можете создавать и использовать простые объекты, но опустимся немного глубже и сделаем нечто большее.

Когда вы пишете что-то вроде a = 3 + 8, откуда целочисленные объекты со значениями 3 и 8 узнают, как реализовать операцию +? Кроме того, откуда a знает, как использовать =, чтобы получить результат? Вы можете воспользоваться этими операторами, применяя *специальные методы* Python (также можно назвать их *магическими методами*). Вам не нужно быть Гэндальфом, чтобы творить магию, эти методы совсем не сложны.

Имена этих методов начинаются с двойных подчеркиваний (__) и заканчиваются ими. Вы уже видели один такой метод: __init__ инициализирует только что созданный объект с помощью описания его класса и любых аргументов, которые были переданы в этот метод.

Предположим, у вас есть простой класс Word и вы хотите написать для него метод equals(), который сравнивает два слова, игнорируя регистр. Так и есть, объект класса Word, содержащий значение 'ha', будет считаться равным другому объекту, который содержит значение 'HA'.

В следующем примере показана наша первая попытка, где мы вызываем обычный метод equals(). self.text — это текстовая строка, которую содержит объект класса Word, метод equals() сравнивает ее с текстовой строкой, содержащейся в объекте word2 (другой объект класса Word):

```
>>> class Word():
...    def __init__(self, text):
```

```
...         self.text = text
...
...     def equals(self, word2):
...         return self.text.lower() == word2.text.lower()
...
```

Далее создадим три объекта `Word` с помощью трех разных текстовых строк:

```
>>> first = Word('ha')
>>> second = Word('HA')
>>> third = Word('eh')
```

Когда строки `'ha'` и `'HA'` сравниваются в нижнем регистре, они должны быть равными:

```
>>> first.equals(second)
True
```

Но строка `'eh'` не совпадет со строкой `'ha'`:

```
>>> first.equals(third)
False
```

Мы определили метод `equals()`, который выполняет преобразование строки в нижний регистр и сравнение. Однако было бы здорово, если бы мы могли просто сказать `first == second`, как в случае встроенных типов Python. Реализуем такую возможность. Мы изменим имя метода `equals()` на особое имя `__eq__()` (вы узнаете, зачем я это сделал, через минуту):

```
>>> class Word():
...     def __init__(self, text):
...         self.text = text
...     def __eq__(self, word2):
...         return self.text.lower() == word2.text.lower()
...
```

Проверим, как это работает:

```
>>> first = Word('ha')
>>> second = Word('HA')
>>> third = Word('eh')
>>> first == second
True
>>> first == third
False
```

Магия! Все, что нам было нужно, — указать особое имя метода для проверки на равенство `__eq__()`. В табл. 6.1 и 6.2 приведены имена самых полезных магических методов.

Таблица 6.1. Магические методы для сравнения

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

Таблица 6.2. Магические методы для вычислений

__add__(self, other)	self + other
__sub__(self, other)	self — other
__mul__(self, other)	self * other
__floordiv__(self, other)	self // other
__truediv__(self, other)	self / other
__mod__(self, other)	self % other
__pow__(self, other)	self ** other

Не обязательно использовать математические операторы вроде `+` (магический метод `__add__()`) и `-` (магический метод `__sub__()`) только для работы с числами. Например, строковые объекты используют `+` для конкатенации и `*` для дуплицирования. Существует множество других методов, задокументированных онлайн по адресу http://bit.ly/pydocs-smn. Наиболее распространенные из них представлены в табл. 6.3.

Таблица 6.3. Другие магические методы

__str__(self)	str(self)
__repr__(self)	repr(self)
__len__(self)	len(self)

Вы можете обнаружить, что, помимо `__init__()`, часто пользуетесь методом `__str__()`. Он нужен для того, чтобы выводить данные на экран. Этот метод используется методами `print()`, `str()` и строками форматирования, о которых вы можете прочитать в главе 7. Интерактивный интерпретатор применяет функцию `__repr__()` для того, чтобы выводить на экран переменные. Если вы не определите хотя бы один из этих методов, то увидите на экране ваш объект, преобразованный в строку по умолчанию:

```
>>> first = Word('ha')
>>> first
<__main__.Word object at 0x1006ba3d0>
>>> print(first)
<__main__.Word object at 0x1006ba3d0>
```

Добавим в класс `Word` методы `__str__()` и `__repr__()`, чтобы он лучше смотрелся:

```
>>> class Word():
...     def __init__(self, text):
...         self.text = text
...     def __eq__(self, word2):
...         return self.text.lower() == word2.text.lower()
...     def __str__(self):
...         return self.text
...     def __repr__(self):
...         return 'Word("'  self.text  '")'
...
>>> first = Word('ha')
>>> first          # используется __repr__
Word("ha")
>>> print(first)   # используется __str__
ha
```

Чтобы узнать о других особых методах, обратитесь к документации Python (http://bit.ly/pydocs-smn).

Композиция

Наследование может сослужить хорошую службу, если вам нужно создать класс-потомок, который ведет себя как родительский класс бóльшую часть времени (когда потомок является предком). Возможность создавать иерархии наследования довольно заманчива, но иногда *композиция* или *агрегирование* (когда *x* имеет *y*) имеет больше смысла. Утка является птицей, но имеет хвост. Хвост не похож на утку, он является частью утки. В следующем примере создадим объекты `bill` и `tail` и предоставим их новому объекту `duck`:

```
>>> class Bill():
...     def __init__(self, description):
...         self.description = description
...
>>> class Tail():
...     def __init__(self, length):
...         self.length = length
...
>>> class Duck():
...     def __init__(self, bill, tail):
...         self.bill = bill
...         self.tail = tail
...     def about(self):
...         print('This duck has a', bill.description, 'bill and a',
...               tail.length, 'tail')
...
>>> tail = Tail('long')
```

```
>>> bill = Bill('wide orange')
>>> duck = Duck(bill, tail)
>>> duck.about()
This duck has a wide orange bill and a long tail
```

Когда лучше использовать классы и объекты, а когда — модули

Рассмотрим несколько рекомендаций, которые помогут вам понять, где лучше разместить свой код — в классе или в модуле.

- Объекты наиболее полезны, когда вам нужно иметь некоторое количество отдельных экземпляров с одинаковым поведением (методами), но различающихся внутренним состоянием (атрибутами).
- Классы, в отличие от модулей, поддерживают наследование.
- Если вам нужен только один объект, модуль подойдет лучше. Независимо от того, сколько обращений к модулю имеется в программе, будет загружена только одна копия. (Программистам на Java и C++: если вы знакомы с книгой Эриха Гаммы «Приемы объектно-ориентированного проектирования. Паттерны проектирования» (Gamma E. *Design Patterns: Elements of Reusable Object-Oriented Software*), можете использовать модули в Python как синглтоны.)
- Если у вас есть несколько переменных, которые содержат разные значения и могут быть переданы как аргументы в несколько функций, лучше всего определить их как классы. Например, вы можете использовать словарь с ключами `size` и `color`, чтобы представить цветное изображение. Вы можете создать разные словари для каждого изображения в программе и передавать их в качестве аргументов в функции `scale()` и `transform()`. По мере добавления новых ключей и функций может начаться путаница. Более последовательно было бы определить класс `Image` с атрибутами `size` или `color` и методами `scale()` и `transform()`. В этом случае все данные и методы для работы с цветными изображениями будут определены в одном месте.
- Используйте простейшее решение задачи. Словарь, список или кортеж проще, компактнее и быстрее, чем модуль, который, в свою очередь, проще, чем класс.

Совет от Гвидо ван Россума: «Избегайте усложнения структур данных. Кортежи лучше объектов (можно воспользоваться именованными кортежами). Предпочитайте простые поля функциям, геттерам и сеттерам. Используйте больше чисел, строк, кортежей, списков, множеств, словарей. Взгляните также на библиотеку collections, особенно на класс `deque`».

Именованные кортежи. Поскольку Гвидо только что упомянул их, а я про них еще не говорил, самое время рассмотреть *именованные кортежи*. Именованный

кортеж — это подкласс кортежей, с помощью которых вы можете получить доступ к значениям по имени (с помощью конструкции .name) и позиции (с помощью конструкции [offset]).

Рассмотрим пример из предыдущего раздела и преобразуем класс Duck в именованный кортеж, сохранив bill и tail как простые строковые аргументы. Функцию namedtuple можно вызвать, передав ей два аргумента:

- имя;
- строку, содержащую имена полей, разделенные пробелами.

Именованные кортежи не поддерживаются Python автоматически, вам понадобится загрузить отдельный модуль для того, чтобы их использовать. Мы сделаем это в первой строке следующего примера:

```
>>> from collections import namedtuple
>>> Duck = namedtuple('Duck', 'bill tail')
>>> duck = Duck('wide orange', 'long')
>>> duck
Duck(bill='wide orange', tail='long')
>>> duck.bill
'wide orange'
>>> duck.tail
'long'
```

Именованный кортеж можно сделать также на основе словаря:

```
>>> parts = {'bill': 'wide orange', 'tail': 'long'}
>>> duck2 = Duck(**parts)
>>> duck2
Duck(bill='wide orange', tail='long')
```

В коде, показанном ранее, обратите внимание на конструкцию **parts. Это *аргумент — ключевое слово*. Он извлекает ключи и значения словаря parts и передает их как аргументы в Duck(). По эффекту это похоже на следующий код:

```
>>> duck2 = Duck(bill = 'wide orange', tail = 'long')
```

Именованные кортежи неизменяемы, но вы можете заменить одно или несколько полей и вернуть другой именованный кортеж:

```
>>> duck3 = duck2._replace(tail='magnificent', bill='crushing')
>>> duck3
Duck(bill='crushing', tail='magnificent')
```

Мы могли бы объявить duck как словарь:

```
>>> duck_dict = {'bill': 'wide orange', 'tail': 'long'}
>>> duck_dict
{'tail': 'long', 'bill': 'wide orange'}
```

Вы можете добавить поля в словарь:

```
>>> duck_dict['color'] = 'green'
>>> duck_dict
{'color': 'green', 'tail': 'long', 'bill': 'wide orange'}
```

Но не в именованный кортеж:

```
>>> duck.color = 'green'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'dict' object has no attribute 'color'
```

Вспомним плюсы использования именованного кортежа.

- Они выглядят и действуют как неизменяемый объект.
- Они более эффективны, чем объекты, с точки зрения времени и занимаемого места.
- Вы можете получить доступ к атрибутам с помощью точки вместо квадратных скобок, характерных для словарей.
- Вы можете использовать их как ключ словаря.

Упражнения

1. Создайте класс, который называется `Thing`, не имеющий содержимого, и выведите его на экран. Затем создайте объект `example` этого класса и также выведите его. Совпадают ли выведенные значения?
2. Создайте новый класс с именем `Thing2` и присвойте его атрибуту `letters` значение `'abc'`. Выведите на экран значение атрибута `letters`.
3. Создайте еще один класс, который, конечно же, называется `Thing3`. В этот раз присвойте значение `'xyz'` атрибуту объекта, который называется `letters`. Выведите на экран значение атрибута `letters`. Понадобилось ли вам создавать объект класса, чтобы сделать это?
4. Создайте класс, который называется `Element`, имеющий атрибуты объекта `name`, `symbol` и `number`. Создайте объект этого класса со значениями `'Hydrogen'`, `'H'` и `1`.
5. Создайте словарь со следующими ключами и значениями: `'name': 'Hydrogen'`, `'symbol': 'H'`, `'number': 1`. Далее создайте объект с именем `hydrogen` класса `Element` с помощью этого словаря.
6. Для класса `Element` определите метод с именем `dump()`, который выводит на экран значения атрибутов объекта (`name`, `symbol` и `number`). Создайте объект `hydrogen` из этого нового определения и используйте метод `dump()`, чтобы вывести на экран его атрибуты.

7. Вызовите функцию `print(hydrogen)`. В определении класса `Element` измените имя метода `dump` на `__str__`, создайте новый объект `hydrogen` и затем снова вызовите метод `print(hydrogen)`.
8. Модифицируйте класс `Element`, сделав атрибуты `name`, `symbol` и `number` закрытыми. Определите для каждого атрибута свойство получателя, возвращающее значение соответствующего атрибута.
9. Определите три класса: `Bear`, `Rabbit` и `Octothorpe`. Для каждого из них определите всего один метод — `eats()`. Он должен возвращать значения `'berries'` (для `Bear`), `'clover'` (для `Rabbit`) или `'campers'` (для `Octothorpe`). Создайте по одному объекту каждого класса и выведите на экран то, что ест указанное животное.
10. Определите три класса: `Laser`, `Claw` и `SmartPhone`. Каждый из них имеет только один метод — `does()`. Он возвращает значения `'disintegrate'` (для `Laser`), `'crush'` (для `Claw`) или `'ring'` (для `SmartPhone`). Далее определите класс `Robot`, который содержит по одному объекту каждого из этих классов. Определите метод `does()` для класса `Robot`, который выводит на экран все, что делают его компоненты.

7 Работаем с данными профессионально

Из этой главы вы узнаете множество приемов приручения данных. Большинство из них касаются встроенных типов данных.

- *Строки* — последовательности символов в кодировке Unicode, используемые для представления текстовых данных.
- *Байты и массивы байтов* — последовательности восьмибитных целых чисел, используемые для представления двоичных данных.

Текстовые строки

Текст является наиболее популярным типом данных для большинства читателей, поэтому мы начнем главу с рассмотрения мощных особенностей текстовых строк в Python.

Unicode

Все текстовые примеры, показанные в книге до этого момента, имели формат ASCII. Этот формат был определен в 1960-х годах, когда компьютеры были размером с холодильник и выполняли вычисления немного лучше последнего. Основной единицей хранения информации был *байт*, который мог хранить 256 уникальных значений в своих 8 *битах*. По разным причинам формат ASCII использовал только 7 бит (128 уникальных значений): 26 символов верхнего регистра, 26 символов нижнего регистра, 10 цифр, некоторые знаки препинания, символы пробела и непечатаемые символы.

К сожалению, в мире существует больше букв, чем предоставляет формат ASCII. Вы могли заказать в кафе хот-дог, но не Gewürztraminer (название этого вина в Германии пишется через ü, а во Франции — без него). Было предпринято множество попыток добавить больше букв и символов, и время от времени вы будете встречать их. Вот некоторые из них:

- Latin-1 или ISO 8859-1;
- Windows code page 1252.

Каждый из этих форматов использует все 8 бит, но даже этого недостаточно, особенно когда вам нужно воспользоваться неевропейскими языками. *Unicode* — это действующий международный стандарт, определяющий символы всех языков мира плюс математические и другие символы.

«Unicode предоставляет уникальный номер каждому символу независимо от платформы, программы и языка» (Консорциум Unicode).

Страница Unicode Code Charts (http://www.unicode.org/charts) содержит ссылки на все определенные на данный момент наборы символов с изображениями. В последней версии (6.2) определяется более 110 000 символов, каждый из которых имеет уникальное имя и идентификационный номер. Символы разбиты на восьмибитные наборы, которые называются *плоскостями*. Первые 256 плоскостей называются *основными многоязычными уровнями*. Обратитесь к странице о плоскостях в «Википедии» (http://bit.ly/unicode-plane), чтобы получить более подробную информацию.

Строки формата Unicode в Python 3

Строки в Python 3 являются строками формата Unicode, а не массивом байтов. Одним разграничением между обычными байтовыми строками и строками в формате Unicode Python 3 значительно отличается от Python 2.

Если вы знаете Unicode ID или название символа, то можете использовать его в строке Python. Вот несколько примеров.

- Символ `\u`, за которым располагаются четыре шестнадцатеричных числа (числа шестнадцатеричной системы счисления, содержащие символы от 0 до 9 и от A до F), определяют символ, находящийся в одной из 256 многоязычных плоскостей Unicode. Первые два числа являются номером плоскости (от 00 до FF), а следующие два — индексом символа внутри плоскости. Плоскость с номером 00 — это старый добрый формат ASCII, и позиции символов в нем такие же, как и в ASCII.
- Для символов более высоких плоскостей нужно больше битов. Управляющая последовательность для них выглядит как `\U`, за которым следуют восемь шестнадцатеричных символов, крайний слева из них должен быть равен 0.
- Для всех символов конструкция `\N{ имя }` позволяет указать символ с помощью его стандартного *имени*. Имена перечислены по адресу http://www.unicode.org/charts/charindex.html.

Модуль `unicodedata` содержит функции, которые преобразуют символы в обоих направлениях:

- `lookup()` принимает не зависящее от регистра имя и возвращает символ Unicode;
- `name()` принимает символ Unicode и возвращает его имя в верхнем регистре.

В следующем примере мы напишем проверочную функцию, которая принимает символ Unicode, ищет его имя, а затем ищет символ, соответствующий полученному имени (он должен совпасть с оригинальным):

```
>>> def unicode_test(value):
...     import unicodedata
...     name = unicodedata.name(value)
...     value2 = unicodedata.lookup(name)
...     print('value="%s", name="%s", value2="%s"' % (value, name, value2))
...
```

Попробуем проверить несколько символов, начиная с простой буквы формата ASCII:

```
>>> unicode_test('A')
value="A", name="LATIN CAPITAL LETTER A", value2="A"
```

Знак препинания, доступный в ASCII:

```
>>> unicode_test('$')
value="$", name="DOLLAR SIGN", value2="$"
```

Символ валюты из Unicode:

```
>>> unicode_test('\u00a2')
value="¢", name="CENT SIGN", value2="¢"
```

Еще один символ валюты из Unicode:

```
>>> unicode_test('\u20ac')
value="€", name="EURO SIGN", value2="€"
```

Единственная проблема, с которой вы можете столкнуться, — это ограничения, накладываемые шрифтом. Ни в одном шрифте нет символов для всех символов Unicode, вместо них будет отображен символ-заполнитель. Например, так выглядит символ Unicode `SNOWMAN`, содержащийся в пиктографических шрифтах:

```
>>> unicode_test('\u2603')
value="☃", name="SNOWMAN", value2="☃"
```

Предположим, мы хотим сохранить в строке слово café. Одно из решений состоит в том, чтобы скопировать его из файла или с сайта и понадеяться, что это сработает:

```
>>> place = 'café'
>>> place
'café'
```

Это сработало, поскольку я скопировал это слово из источника, использующего кодировку UTF-8 (с которой вы познакомитесь далее), и вставил его.

Как же нам указать, что последний символ — это «é»? Если вы посмотрите на индекс символа «E», вы увидите, что имя `E WITH ACUTE, LATIN SMALL LETTER` имеет индекс

00E9. Рассмотрим функции `name()` и `lookup()`, с которыми мы только что работали. Сначала передадим код символа, чтобы получить его имя:

```
>>> unicodedata.name('\u00e9')
'LATIN SMALL LETTER E WITH ACUTE'
```

Теперь найдем код для заданного имени:

```
>>> unicodedata.lookup('E WITH ACUTE, LATIN SMALL LETTER')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: "undefined character name 'E WITH ACUTE, LATIN SMALL LETTER'"
```

Имена, перечисленные в списке Unicode Character Name Index, были переформатированы для удобства отображения. Для того чтобы преобразовать их в настоящие имена символов Unicode (которые используются в Python), удалите запятую и переместите ту часть имени, которая находится после нее, в самое начало. Соответственно, в нашем примере E WITH ACUTE, LATIN SMALL LETTER нужно изменить на LATIN SMALL LETTER E WITH ACUTE:

```
>>> unicodedata.lookup('LATIN SMALL LETTER E WITH ACUTE')
'é'
```

Теперь мы можем использовать символ «é» как с помощью кода, так и с помощью имени:

```
>>> place = 'caf\u00e9'
>>> place
'café'
>>> place = 'caf\N{LATIN SMALL LETTER E WITH ACUTE}'
>>> place
'café'
```

В предыдущем сниппете вы вставили символ «é» непосредственно в строку, но мы также можем собрать строку из составляющих:

```
>>> u_umlaut = '\N{LATIN SMALL LETTER U WITH DIAERESIS}'
>>> u_umlaut
'ú'
>>> drink = 'Gew' + u_umlaut + 'rztraminer'
>>> print('Now I can finally have my', drink, 'in a', place)
Now I can finally have my Gewúrztraminer in a café
```

Строковая функция `len` считает количество символов в кодировке Unicode, а не байты:

```
>>> len('$')
1
>>> len('\U0001f47b')
1
```

Кодирование и декодирование с помощью кодировки UTF-8

Вам не нужно волноваться о том, как Python хранит каждый символ Unicode, когда вы выполняете обычную обработку строки.

Но когда вы обмениваетесь данными с внешним миром, вам может понадобиться следующее:

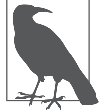
- способ *закодировать* строку с помощью байтов;
- способ *декодировать* байты обратно в строку.

Если бы в Unicode было менее 64 000 символов, мы могли бы хранить ID каждого из них в двух байтах. К сожалению, символов больше. Мы могли бы кодировать каждый ID с помощью трех или четырех байтов, но это увеличило бы объем памяти и дискового пространства, необходимый для обычных текстовых строк, в три или четыре раза.

Кен Томпсон (Ken Thompson) и Роб Пайк (Rob Pike), чьи имена будут знакомы разработчикам на Unix, разработали UTF-8 — динамическую схему кодирования — однажды вечером на салфетке в одной из столовых Нью-Джерси. Она использует для символа Unicode от одного до четырех байтов:

- один байт для ASCII;
- два байта для большинства языков, основанных на латинице (но не кириллице);
- три байта для остальных простых языков;
- четыре байта для остальных языков, включая некоторые азиатские языки и символы.

UTF-8 — это стандартная текстовая кодировка для Python, Linux и HTML. Она охватывает множество символов, работает быстро и хорошо. Если вы используете кодировку UTF-8 в своем коде, жизнь станет гораздо проще, чем в том случае, если будете скакать от одной кодировки к другой.



Если вы создаете строку Python путем копирования символов из другого источника вроде веб-страницы и их вставки, убедитесь, что источник был закодирован с помощью UTF-8. Очень часто может оказаться, что текст был зашифрован с помощью кодировок Latin-1 или Windows 1252, что при копировании в строку Python вызовет генерацию исключений из-за некорректной последовательности байтов.

Кодирование

Вы кодируете строку байтами. Первый аргумент строковой функции `encode()` — это имя кодировки. Возможные варианты представлены в табл. 7.1.

Таблица 7.1. Кодировки

ascii	Старая добрая семибитная кодировка ASCII
utf-8	Восьмибитная кодировка переменной длины, самый предпочтительный вариант в большинстве случаев
latin-1	Также известна как ISO 8859-1
cp-1252	Стандартная кодировка Windows
unicode-escape	Буквенный формат Python Unicode, выглядит как \uxxxx или \Uxxxxxxxx

С помощью кодировки UTF-8 вы можете закодировать все что угодно. Присвоим строку Unicode `'\u2603'` переменной `snowman`:

```
>>> snowman = '\u2603'
```

`snowman` — это строка Python Unicode, содержащая один символ независимо от того, сколько байтов может потребоваться для того, чтобы сохранить ее:

```
>>> len(snowman)
1
```

Теперь закодируем этот символ последовательностью байтов:

```
>>> ds = snowman.encode('utf-8')
```

Как я упоминал ранее, кодировка UTF-8 имеет переменную длину. В этом случае было использовано три байта для того, чтобы закодировать один символ `snowman`:

```
>>> len(ds)
3
>>> ds
b'\xe2\x98\x83'
```

Функция `len()` возвращает число байтов (3), поскольку `ds` является переменной `bytes`.

Вы можете использовать другие кодировки, не только UTF-8, но будете получать ошибки, если строка Unicode не сможет быть обработана другой кодировкой. Например, если вы используете кодировку `ascii`, у вас ничего не выйдет, если только вы не предоставите строку, состоящую из корректных символов ASCII:

```
>>> ds = snowman.encode('ascii')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
UnicodeEncodeError: 'ascii' codec can't encode character '\u2603'
in position 0: ordinal not in range(128)
```

Функция `encode()` принимает второй аргумент, который помогает вам избежать возникновения исключений, связанных с кодировкой. Его значение по умолчанию, как вы можете увидеть в предыдущем примере, равно `'strict'`; при таком значении

наблюдается исключение UnicodeEncodeError, если встречается символ, не входящий в кодировку ASCII. Существуют и другие кодировки. Используйте значение 'ignore', чтобы опустить все, что не может быть закодировано:

```
>>> snowman.encode('ascii', 'ignore')
b''
```

Используйте значение 'replace', чтобы заменить неизвестные символы символами ?:

```
>>> snowman.encode('ascii', 'replace')
b'?'
```

Используйте значение 'backslashreplace', чтобы создать строку, содержащую символы Python Unicode вроде unicode-escape:

```
>>> snowman.encode('ascii', 'backslashreplace')
b'\\u2603'
```

Вы можете использовать этот вариант, если вам нужна печатаемая версия управляющей последовательности Unicode.

В следующем примере создаются строки символьных сущностей, которые вы можете встретить на веб-страницах:

```
>>> snowman.encode('ascii', 'xmlcharrefreplace')
b'&#9731;'
```

Декодирование

Мы декодируем байтовые строки в строки Unicode. Когда мы получаем текст из какого-то внешнего источника (файлы, базы данных, сайты, сетевые API и т. д.), он закодирован в виде байтовой строки. Идея заключается в том, чтобы знать, какая кодировка была использована, чтобы мы могли ее декодировать и получить строку Unicode.

Проблема в следующем: никакая часть байтовой строки не говорит нам о том, какая была использована кодировка. Я уже упоминал опасности копирования/вставки с сайтов. Вы, возможно, посещали сайты, содержащие странные символы в том месте, где должны быть простые символы ASCII.

Создадим строку Unicode, которая называется place и имеет значение 'café':

```
>>> place = 'caf\u00e9'
>>> place
'café'
>>> type(place)
<class 'str'>
```

Закодируем ее в формат UTF-8 с помощью переменной bytes, которая называется place_bytes:

```
>>> place_bytes = place.encode('utf-8')
```

```
>>> place_bytes
b'caf\xc3\xa9'
>>> type(place_bytes)
<class 'bytes'>
```

Обратите внимание на то, что переменная `place_bytes` содержит пять байтов. Первые три похожи на ASCII (преимущество UTF-8), а последние два кодируют символ «é». Теперь декодируем эту байтовую строку обратно в строку Unicode:

```
>>> place2 = place_bytes.decode('utf-8')
>>> place2
'café'
```

Это сработало, поскольку мы закодировали и декодировали строку с помощью кодировки UTF-8. Что, если бы мы указали декодировать ее с помощью какой-нибудь другой кодировки?

```
>>> place3 = place_bytes.decode('ascii')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
UnicodeDecodeError: 'ascii' codec can't decode byte 0xc3 in position 3:
ordinal not in range(128)
```

Декодер ASCII сгенерировал исключение, поскольку байтовое значение `0xc3` некорректно в ASCII. Существуют и другие восьмибитные кодировки, где значения между 128 (80 в шестнадцатеричной системе) и 255 (FF в шестнадцатеричной системе) корректны, но не совпадают со значениями UTF-8:

```
>>> place4 = place_bytes.decode('latin-1')
>>> place4
'cafÃ©'
>>> place5 = place_bytes.decode('windows-1252')
>>> place5
'cafÃ©'
```

Ох.

Мораль этой истории — используйте кодировку UTF-8 всюду, где это возможно. Она работает, она поддерживается везде, вы можете с ее помощью выразить любой символ Unicode и быстро закодировать и декодировать.

Подробная информация

Если вы хотите узнать больше, вам могут помочь следующие ссылки:

- Unicode HOWTO (http://bit.ly/unicode-howto);
- Pragmatic Unicode (http://bit.ly/pragmatic-uni);
- The Absolute Minimum Every Software Developer Absolutely, Positively Must Know About Unicode and Character Sets (No Excuses!) (http://bit.ly/jspolsky).

Формат

До этого момента мы просто игнорировали форматирование текста. В главе 2 были показаны несколько функций для выравнивания строк, а в примерах кода использовалась простая функция `print()` или даже вывод информации на экран поручался интерактивному интерпретатору. Но теперь мы рассмотрим, как *интерполировать* данные в строки — другими словами, разместить значения внутри строк, — применяя разные форматы. Вы можете использовать эту возможность, чтобы создавать отчеты и другие документы, для которых нужно задать определенный внешний вид.

Python предлагает два способа форматирования строк, их часто называют *старым стилем* и *новым стилем*. Оба стиля поддерживаются Python 2 и 3 (новый стиль появился в Python 2.6). Старый стиль проще, поэтому мы начнем с него.

Старый стиль с символом %

Старый стиль форматирования строк имеет форму *`строка % данные`*. Внутри строки находятся интерполяционные последовательности. В табл. 7.2 показано, что самая простая последовательность — это символ `%`, за которым следует буква, представляющая тип данных, который должен быть отформатирован.

Таблица 7.2. Типы преобразования

%s	Строка
%d	Целое число в десятичной системе счисления
%x	Целое число в шестнадцатеричной системе счисления
%o	Целое число в восьмеричной системе счисления
%f	Число с плавающей точкой в десятичной системе счисления
%e	Число с плавающей точкой в шестнадцатеричной системе счисления
%g	Число с плавающей точкой в восьмеричной системе счисления
%%	Символ %

Далее мы рассмотрим несколько примеров. Сначала целое число:

```
>>> '%s' % 42
'42'
>>> '%d' % 42
'42'
>>> '%x' % 42
'2a'
>>> '%o' % 42
'52'
```

Число с плавающей точкой:

```
>>> '%s' % 7.03
'7.03'
>>> '%f' % 7.03
'7.030000'
>>> '%e' % 7.03
'7.030000e+00'
>>> '%g' % 7.03
'7.03'
```

Целое число и символ %:

```
>>> '%d%%' % 100
'100%'
```

Интерполяция некоторых строк и целых чисел:

```
>>> actor = 'Richard Gere'
>>> cat = 'Chester'
>>> weight = 28
>>> "My wife's favorite actor is %s" % actor
"My wife's favorite actor is Richard Gere"
>>> "Our cat %s weighs %s pounds" % (cat, weight)
'Our cat Chester weighs 28 pounds'
```

Последовательность %s внутри строки означает, что в нее нужно интерполировать строку. Количество использованных символов % должно совпадать с количеством объектов, которые располагаются после %. Один объект вроде actor располагается сразу после символа %. Если таких объектов несколько, они должны быть сгруппированы в кортеж (нужно окружить их скобками и разделить запятыми) вроде (cat, weight).

Несмотря на то что переменная weight целочисленная, последовательность %s внутри строки преобразует ее в строку.

Вы можете добавить другие значения между % и определением типа, чтобы указать минимальную и максимальную ширину, выравнивание и заполнение символами.

Определим несколько переменных: целочисленную n, число с плавающей точкой f и строку s:

```
>>> n = 42
>>> f = 7.03
>>> s = 'string cheese'
```

Отформатируем их, используя ширину по умолчанию:

```
>>> '%d %f %s' % (n, f, s)
'42 7.030000 string cheese'
```

Установим минимальную длину поля, равную 10 символам, для каждой переменной и выровняем их по правому краю, заполняя неиспользованное место пробелами:

```
>>> '%10d %10f %10s' % (n, f, s)
'        42   7.030000 string cheese'
```

Используем ту же ширину поля, но выравнивание будет по левому краю:

```
>>> '%-10d %-10f %-10s' % (n, f, s)
'42         7.030000   string cheese'
```

В этот раз укажем ту же длину поля, но максимальное количество символов будет равно 4, выровняем все по правому краю. Такая настройка обрезает строку и ограничивает число с плавающей точкой четырьмя цифрами после десятичной запятой:

```
>>> '%10.4d %10.4f %10.4s' % (n, f, s)
'      0042     7.0300       stri'
```

То же самое, но выравнивание по правому краю:

```
>>> '%.4d %.4f %.4s' % (n, f, s)
'0042 7.0300 stri'
```

Наконец, получим длину полей из аргументов, вместо того чтобы жестко ее закодировать:

```
>>> '%*.*d %*.*f %*.*s' % (10, 4, n, 10, 4, f, 10, 4, s)
'      0042     7.0300       stri'
```

Новый стиль форматирования с помощью символов {} и функции format

Старый стиль форматирования все еще поддерживается. В Python 2, который остановился на версии 2.7, он будет поддерживаться всегда. Но если вы работаете с Python 3, рекомендуется применять новый стиль форматирования.

Простейший пример его использования показан здесь:

```
>>> '{} {} {}'.format(n, f, s)
'42 7.03 string cheese'
```

Аргументы старого стиля нужно предоставлять в порядке появления их заполнителей с символами `%` в оригинальной строке. С помощью нового стиля вы можете указывать любой порядок:

```
>>> '{2} {0} {1}'.format(f, s, n)
'42 7.03 string cheese'
```

Значение `0` относится к первому аргументу, `f`, `1` относится к строке `s`, а `2` — к последнему аргументу, целому числу `n`.

Аргументы могут являться словарем или именованными аргументами, а спецификаторы могут включать их имена:

```
>>> '{n} {f} {s}'.format(n=42, f=7.03, s='string cheese')
'42 7.03 string cheese'
```

В следующем примере попробуем объединить три наших значения в словарь, который выглядит так:

```
>>> d = {'n': 42, 'f': 7.03, 's': 'string cheese'}
```

В следующем примере `{0}` подразумевает весь словарь, а `{1}` — строку `'other'`, которая следует за словарем:

```
>>> '{0[n]} {0[f]} {0[s]} {1}'.format(d, 'other')
'42 7.03 string cheese other'
```

В этих примерах аргументы выводятся на экран с форматированием по умолчанию. Старый стиль позволяет указать спецификатор типа после символа `%`, а новый стиль — после `:`. Начнем с аргументов позиционирования:

```
>>> '{0:d} {1:f} {2:s}'.format(n, f, s)
'42 7.030000 string cheese'
```

В этом примере мы используем те же значения, но для именованных аргументов:

```
>>> '{n:d} {f:f} {s:s}'.format(n=42, f=7.03, s='string cheese')
'42 7.030000 string cheese'
```

Другие возможности (минимальная длина поля, максимальная ширина символов, смещение и т. д.) также поддерживаются.

Минимальная длина поля — 10, выравнивание по правому краю (по умолчанию):

```
>>> '{0:10d} {1:10f} {2:10s}'.format(n, f, s)
'        42   7.030000 string cheese'
```

То же, что и в предыдущем примере, но символы `>` делают выравнивание по правому краю более явным:

```
>>> '{0:>10d} {1:>10f} {2:>10s}'.format(n, f, s)
'        42   7.030000 string cheese'
```

Минимальная длина поля — 10, выравнивание по левому краю:

```
>>> '{0:<10d} {1:<10f} {2:<10s}'.format(n, f, s)
'42         7.030000   string cheese'
```

Минимальная длина поля — 10, выравнивание по центру:

```
>>> '{0:^10d} {1:^10f} {2:^10s}'.format(n, f, s)
'    42      7.030000  string cheese'
```

Есть один момент, который отличает старый стиль от нового: значение *точности* (после десятичной запятой) все еще означает количество цифр после десятичной

запятой для дробных чисел и максимальное число символов строки, но вы не можете использовать его для целых чисел:

```
>>> '{0:>10.4d} {1:>10.4f} {2:10.4s}'.format(n, f, s)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: Precision not allowed in integer format specifier
>>> '{0:>10d} {1:>10.4f} {2:>10.4s}'.format(n, f, s)
'        42     7.0300       stri'
```

Последняя опция — это *символ-заполнитель*. Если вы хотите заполнить поле вывода чем-то кроме пробелов, разместите этот символ сразу после двоеточия, но перед символами выравнивания (<, >, ^) или спецификатором ширины:

```
>>> '{0:!^20s}'.format('BIG SALE')
'!!!!!!BIG SALE!!!!!!'
```

Совпадение с регулярными выражениями

В главе 2 немного рассматривались операции со строками. Вооружившись этой промежуточной информацией, вы, возможно, использовали простые шаблоны в командной строке, содержащие символ подстановки, вроде `ls *.py`, что означает «перечислить все имена файлов, заканчивающиеся на `.py`».

Пришло время рассмотреть более сложный механизм проверки на совпадение с шаблоном — *регулярные выражения*. Этот механизм поставляется в стандартном модуле `re`, который мы импортируем. Вы определяете строковый *шаблон*, совпадения для которого вам нужно найти, и строку-*источник*, в которой следует выполнить поиск. Простой пример использования выглядит так:

```
result = re.match('You', 'Young Frankenstein')
```

В этом примере строка `'You'` является *шаблоном*, а `'Young Frankenstein'` — *источником*, строкой, которую вы хотите проверить. Функция `match()` проверяет, начинается ли *источник* с *шаблона*.

Для более сложных проверок вам нужно *скомпилировать* шаблон, чтобы ускорить поиск:

```
youpattern = re.compile('You')
```

Далее вы можете выполнить проверку с помощью скомпилированного шаблона:

```
result = youpattern.match('Young Frankenstein')
```

Функция `match()` — это не единственный способ сравнить шаблон и источник, существует еще несколько методов.

- `search()` возвращает первое совпадение, если таковое имеется.
- `findall()` возвращает список всех непересекающихся совпадений, если таковые имеются.

- `split()` разбивает источник на совпадения с шаблоном и возвращает список всех фрагментов строки.
- `sub()` принимает *аргумент для замены* и заменяет все части источника, совпавшие с шаблоном, на значение этого аргумента.

Точное совпадение с помощью функции match()

Начинается ли строка `'Young Frankenstein'` со слова `'You'`? Рассмотрим пример кода с комментариями:

```
>>> import re
>>> source = 'Young Frankenstein'
>>> m = re.match('You', source)  # функция начинает работать с начала источника
>>> if m:  # функция возвращает объект; делайте это, чтобы увидеть, что совпало
...     print(m.group())
...
You
>>> m = re.match('^You', source) # якорь в начале строки делает то же самое
>>> if m:
...     print(m.group())
...
You
```

Как насчет `'Frank'`?

```
>>> m = re.match('Frank', source)
>>> if m:
...     print(m.group())
...
```

В этот раз функция `match()` не вернула ничего, и оператор `if` не запустил оператор `print`. Как я говорил ранее, функция `match()` работает только в том случае, если шаблон находится в начале источника. Но функция `search()` ищет шаблон в любом месте источника:

```
>>> m = re.search('Frank', source)
>>> if m:
...      print(m.group())
...
Frank
```

Изменим шаблон:

```
>>> m = re.match('.*Frank', source)
>>> if m:  # match returns an object
...     print(m.group())
...
Young Frank
```

Кратко объясню, как работает наш новый шаблон:

- символ . означает любой символ;
- символ * означает любое количество предыдущих элементов. Если объединить символы .*, они будут означать любое количество символов (даже ноль);
- 'Frank' — это фраза, которую мы хотим найти в любом месте строки.

Функция match() вернула строку, в которой нашлось совпадение с шаблоном .*Frank: 'Young Frank'.

Первое совпадение, найденное с помощью функции search()

Вы можете использовать функцию search(), чтобы найти шаблон 'Frank' в любом месте строки-источника 'Young Frankenstein', не прибегая к использованию символа подстановки .*:

```
>>> m = re.search('Frank', source)
>>> if m:  # функция search возвращает объект
...     print(m.group())
...
Frank
```

Ищем все совпадения с помощью функции findall()

В предыдущих примерах мы искали только одно совпадение. Но что, если вы хотите узнать, сколько раз строка, содержащая один символ n, встречается в строке-источнике?

```
>>> m = re.findall('n', source)
>>> m   # findall returns a list
['n', 'n', 'n', 'n']
>>> print('Found', len(m), 'matches')
Found 4 matches
```

Как насчет строки 'n', за которой следует любой символ?

```
>>> m = re.findall('n.', source)
>>> m
['ng', 'nk', 'ns']
```

Обратите внимание на то, что в совпадения не была записана последняя строка 'n'. Нам нужно сказать, что символ после 'n' является опциональным, с помощью конструкции ?:

```
>>> m = re.findall('n.?', source)
>>> m
['ng', 'nk', 'ns', 'n']
```

Разбиваем совпадения с помощью функции split()

В следующем примере показано, как разбить строку на список с помощью шаблона, а не простой строки (как это делает метод `split()`):

```
>>> m = re.split('n', source)
>>> m    # функция split возвращает список
['You', 'g Fra', 'ke', 'stei', '']
```

Заменяем совпадения с помощью функции sub()

Этот метод похож на метод `replace()`, но он ищет совпадения с шаблонами, а не простые строки:

```
>>> m = re.sub('n', '?', source)
>>> m   # sub returns a string
'You?g Fra?ke?stei?'
```

Шаблоны: специальные символы

Многие описания регулярных выражений начинаются с деталей, касающихся того, как их определить. Я считаю, что это ошибка. Язык регулярных выражений не так уж мал сам по себе, слишком много деталей должно вместиться в вашу голову одновременно. Они используют так много знаков препинания, что это выглядит так, будто персонажи мультиков ругаются.

Теперь, когда вы знаете о нужных функциях (`match()`, `search()`, `findall()` и `sub()`), рассмотрим детали построения регулярных выражений. Создаваемые вами шаблоны подойдут к любой из этих функций.

Самые простые знаки вы уже видели.

- Совпадения с любыми неспециальными символами.
- Любой отдельный символ, кроме `\n`, — это символ `.`.
- Любое число, включая 0, — это символ `*`.
- Опциональное значение (0 или 1) — это символ `?`.

Специальные символы показаны в табл. 7.3.

Таблица 7.3. Специальные символы

Шаблон	Совпадения
\d	Цифровой символ
\D	Нецифровой символ
\w	Буквенный или цифровой символ или знак подчеркивания
\W	Любой символ, кроме буквенного или цифрового символа или знака подчеркивания
\s	Пробельный символ
\S	Непробельный символ
\b	Граница слова
\B	Не граница слова

Модуль Python `string` содержит заранее определенные строковые константы, которые мы можем использовать для тестирования. Мы воспользуемся константой `printable`, которая содержит 100 печатаемых символов ASCII, включая буквы в обоих регистрах, цифры, пробелы и знаки пунктуации:

```
>>> import string
>>> printable = string.printable
>>> len(printable)
100
>>> printable[0:50]
'0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMN'
>>> printable[50:]
'OPQRSTUVWXYZ!"#$%&\'()*+,-./:;<=>?@[\\]^_`{|}~ \t\n\r\x0b\x0c'
```

Какие символы строки `printable` являются цифрами?

```
>>> re.findall('\d', printable)
['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']
```

Какие символы являются цифрами, буквами и нижним подчеркиванием?

```
>>> re.findall('\w', printable)
['0', '1', '2', '3', '4', '5', '6', '7', '8', '9', 'a', 'b',
'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n',
'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z',
'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L',
'M', 'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X',
'Y', 'Z', '_']
```

Какие символы являются пробелами?

```
>>> re.findall('\s', printable)
[' ', '\t', '\n', '\r', '\x0b', '\x0c']
```

Регулярные выражения не ограничиваются символами ASCII. Шаблон `\d` совпадет со всем, что в кодировке Unicode считается цифрой, а не только с символами ASCII от `0` до 9. Добавим две буквы в нижнем регистре не из ASCII из `FileFormat.info`.

В этой проверке мы добавим туда следующие символы:

- три буквы ASCII;
- три знака препинания, которые не должны совпасть с шаблоном `\w`;
- символ Unicode `LATIN SMALL LETTER E WITH CIRCUMFLEX` (`\u00ea`);
- символ Unicode `LATIN SMALL LETTER E WITH BREVE` (`\u0115`):

```
>>> x = 'abc' + '-/*' + '\u00ea' + '\u0115'
```

Как и ожидалось, этот шаблон нашел только буквы:

```
>>> re.findall('\w', x)
['a', 'b', 'c', 'к', 'ĕ']
```

Шаблоны: использование спецификаторов

Теперь сделаем пиццу из знаков препинания, используя основные спецификаторы шаблонов для регулярных выражений, показанные в табл. 7.4.

В этой таблице *expr* и другие слова, выделенные курсивом, означают любое корректное регулярное выражение.

Таблица 7.4. Спецификаторы шаблонов

Шаблон	**Совпадения**
abc	Буквосочетание abc
(expr)	expr
expr1 | expr2	expr1 или expr2
	Любой символ, кроме \n
^	Начало строки источника
$	Конец строки источника
prev ?	Ноль или одно включение prev
prev *	Ноль или больше включений prev, максимальное количество
prev *?	Ноль или больше включений prev, минимальное количество
prev +	Одно или больше включений prev, максимальное количество
prev +?	Одно или больше включений prev, минимальное количество
prev { m }	m последовательных включений prev
prev { m, n }	От m до n последовательных включений prev, максимальное количество
prev { m, n }?	От m до n последовательных включений prev, минимальное количество
[abc]	a, или b, или c (аналогично a|b|c)
[^abc]	Не (a, или b, или c)
prev (?= next)	prev, если за ним следует next
prev (? ! next)	prev, если за ним не следует next
(?<=prev) next	next, если перед ним находится prev
(?<! prev) next	next, если перед ним не находится prev

У вас могло зарябить в глазах при попытке прочесть эти примеры. Для начала определим строку-источник:

```
>>> source = '''I wish I may, I wish I might
... Have a dish of fish tonight.'''
```

Найдем во всем тексте строку `'wish'`:

```
>>> re.findall('wish', source)
['wish', 'wish']
```

Далее найдем во всем тексте строки 'wish' или 'fish':

```
>>> re.findall('wish|fish', source)
['wish', 'wish', 'fish']
```

Найдем строку 'wish' в начале текста:

```
>>> re.findall('^wish', source)
[]
```

Найдем строку 'I wish' в начале текста:

```
>>> re.findall('^I wish', source)
['I wish']
```

Найдем строку 'fish' в конце текста:

```
>>> re.findall('fish$', source)
[]
```

Наконец, найдем строку 'fish tonight.$' в конце текста:

```
>>> re.findall('fish tonight.$', source)
['fish tonight.']
```

Символы ^ и $ называются *якорями*: с помощью якоря ^ выполняется поиск в начале строки, а с помощью якоря $ — в конце. Сочетание .$ совпадает с любым символом в конце строки, включая точку, поэтому выражение сработало. Для обеспечения большей точности нужно создать управляющую последовательность, чтобы найти именно точку:

```
>>> re.findall('fish tonight\.$', source)
['fish tonight.']
```

Начнем с поиска символов w или f, за которым следует буквосочетание ish:

```
>>> re.findall('[wf]ish', source)
['wish', 'wish', 'fish']
```

Найдем одно или несколько сочетаний символов w, s и h:

```
>>> re.findall('[wsh]+', source)
['w', 'sh', 'w', 'sh', 'h', 'sh', 'sh', 'h']
```

Найдем сочетание ght, за которым следует любой символ, кроме буквенного или цифрового символа или знака подчеркивания:

```
>>> re.findall('ght\W', source)
['ght\n', 'ght.']
```

Найдем символ I, за которым следует сочетание wish:

```
>>> re.findall('I (?=wish)', source)
['I ', 'I ']
```

И наконец, сочетание `wish`, перед которым находится `I`:

```
>>> re.findall('(?<=I) wish', source)
[' wish', ' wish']
```

Существует несколько ситуаций, в которых правила шаблонов регулярных выражений конфликтуют с правилами для строк Python. Следующий шаблон должен совпасть с любым словом, которое начинается с `fish`:

```
>>> re.findall('\bfish', source)
[]
```

Почему этого не произошло? Как мы говорили в главе 2, Python использует специальные *управляющие последовательности* для строк. Например, `\b` для строки означает «возврат на шаг», но в мини-языке регулярных выражений эта последовательность означает начало слова. Избегайте случайного применения управляющих последовательностей, используя *неформатированные строки* Python, когда определяете строку регулярного выражения. Всегда размещайте символ `r` перед строкой шаблона регулярного выражения, и управляющие последовательности Python будут отключены, как показано здесь:

```
>>> re.findall(r'\bfish', source)
['fish']
```

Шаблоны: указываем способ вывода совпадения

При использовании функций `match()` или `search()` все совпадения можно получить из объекта результата `m`, вызвав функцию `m.group()`. Если вы заключите шаблон в круглые скобки, совпадения будут сохранены в отдельную группу и кортеж, состоящий из них, окажется доступен благодаря вызову `m.groups()`, как показано здесь:

```
>>> m = re.search(r'(. dish\b).*(\bfish)', source)
>>> m.group()
'a dish of fish'
>>> m.groups()
('a dish', 'fish')
```

Если вы используете этот шаблон `(?P< name > expr )`, он совпадет с выражением `expr`, сохраняя совпадение в группе `name`:

```
>>> m = re.search(r'(?P<DISH>. dish\b).*(?P<FISH>\bfish)', source)
>>> m.group()
'a dish of fish'
>>> m.groups()
('a dish', 'fish')
>>> m.group('DISH')
'a dish'
>>> m.group('FISH')
'fish'
```

Бинарные данные

Работать с текстовыми данными может быть трудно, но работать с бинарными может быть... интересно. Вам нужно знать о таких концепциях, как *порядок следования байтов* (как процессор вашего компьютера разбивает данные на байты) и *знаковые биты* для целых чисел. Вам может понадобиться закопаться в бинарные форматы файлов или сетевых пакетов, чтобы извлечь или даже изменить данные. В этом разделе я покажу вам основы работы с бинарными данными в Python.

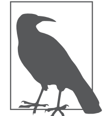
bytes и bytearray

В Python 3 появились следующие последовательности восьмибитных целых чисел, имеющих возможные значения от 0 до 255. Они могут быть двух типов:

- bytes неизменяем, как кортеж байтов;
- bytearray изменяем, как список байтов.

Начнем мы с создания списка с именем blist и в следующем примере создадим переменную типа bytes с именем the_bytes и переменную bytearray с именем the_byte_array:

```
>> blist = [1, 2, 3, 255]
>>> the_bytes = bytes(blist)
>>> the_bytes
b'\x01\x02\x03\xff'
>>> the_byte_array = bytearray(blist)
>>> the_byte_array
bytearray(b'\x01\x02\x03\xff')
```



Представление значения типа bytes начинается с символа b и кавычки, за которыми следуют шестнадцатеричные последовательности вроде \x02 или символы ASCII, заканчивается конструкция соответствующим символом кавычки. Python преобразует шестнадцатеричные последовательности или символы ASCII в маленькие целые числа, но показывает байтовые значения, которые корректно записаны с точки зрения кодировки ASCII:

```
>>> b'\x61'
b'a'
>>> b'\x01abc\xff'
b'\x01abc\xff'
```

В следующем примере показано, что вы не можете изменить переменную типа bytes:

```
>>> the_bytes[1] = 127
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
TypeError: 'bytes' object does not support item assignment
```

Но переменная типа `bytearray` слишком мягкая и легко изменяемая:

```
>>> the_byte_array = bytearray(blist)
>>> the_byte_array
bytearray(b'\x01\x02\x03\xff')
>>> the_byte_array[1] = 127
>>> the_byte_array
bytearray(b'\x01\x7f\x03\xff')
```

Каждая из этих переменных может содержать результат, состоящий из 256 элементов, имеющих значения от 0 до 255:

```
>>> the_bytes = bytes(range(0, 256))
>>> the_byte_array = bytearray(range(0, 256))
```

При выводе на экран содержимого переменных типа `bytes` или `bytearray` Python использует формат `\x xx` для непечатаемых байтов и их эквиваленты ASCII для печатаемых (плюс некоторых распространенных управляющих последовательностей вроде `\n` вместо `\x0a`). Так выглядит на экране представление значения переменной `the_bytes` (переформатированное вручную для того, чтобы показать по 16 байт на строку):

```
>>> the_bytes
b'\x00\x01\x02\x03\x04\x05\x06\x07\x08\t\n\x0b\x0c\r\x0e\x0f
\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f
!"#$%&\'()*+,-./
0123456789:;<=>?
@ABCDEFGHIJKLMNO
PQRSTUVWXYZ[\\]^_
`abcdefghijklmno
pqrstuvwxyz{|}~\x7f
\x80\x81\x82\x83\x84\x85\x86\x87\x88\x89\x8a\x8b\x8c\x8d\x8e\x8f
\x90\x91\x92\x93\x94\x95\x96\x97\x98\x99\x9a\x9b\x9c\x9d\x9e\x9f
\xa0\xa1\xa2\xa3\xa4\xa5\xa6\xa7\xa8\xa9\xaa\xab\xac\xad\xae\xaf
\xb0\xb1\xb2\xb3\xb4\xb5\xb6\xb7\xb8\xb9\xba\xbb\xbc\xbd\xbe\xbf
\xc0\xc1\xc2\xc3\xc4\xc5\xc6\xc7\xc8\xc9\xca\xcb\xcc\xcd\xce\xcf
\xd0\xd1\xd2\xd3\xd4\xd5\xd6\xd7\xd8\xd9\xda\xdb\xdc\xdd\xde\xdf
\xe0\xe1\xe2\xe3\xe4\xe5\xe6\xe7\xe8\xe9\xea\xeb\xec\xed\xee\xef
\xf0\xf1\xf2\xf3\xf4\xf5\xf6\xf7\xf8\xf9\xfa\xfb\xfc\xfd\xfe\xff'
```

Это может выглядеть запутанно, поскольку перед нами байты (маленькие целые числа), а не символы.

Преобразуем бинарные данные с помощью модуля struct

Как вы уже видели, в Python содержится множество инструментов для манипулирования текстом. Инструменты для бинарных данных гораздо менее распространены. Стандартная библиотека содержит модуль struct, который обрабатывает данные аналогично *структурам* в C или C++. С помощью этого модуля вы можете преобразовать бинарные данные в структуры данных Python и наоборот.

Посмотрим, как он работает с данными из файла с расширением PNG — распространенного формата изображений, который вы можете встретить наряду с GIF и JPEG. Мы напишем небольшую программу, которая извлекает ширину и высоту изображения из фрагмента данных PNG.

Используем логотип издательства O'Reilly — изображение маленького долгопята с глазками-бусинами (рис. 7.1).

Рис. 7.1. Долгопят от издательства O'Reilly

Файл этого изображения с расширением PNG доступен в «Википедии». Мы не будем рассматривать чтение файла вплоть до главы 8, поэтому я загрузил этот файл, написал небольшую программу, которая выводит его значения как байты, и просто напечатал значения первых 30 байт как значения переменной типа bytes по имени data для примера, который следует далее. (Спецификация формата PNG предполагает, что ширина и высота хранятся в первых 24 байтах, поэтому нам пока что больше данных и не нужно.)

```
>>> import struct
>>> valid_png_header = b'\x89PNG\r\n\x1a\n'
>>> data = b'\x89PNG\r\n\x1a\n\x00\x00\x00\rIHDR' + \
...     b'\x00\x00\x00\x9a\x00\x00\x00\x8d\x08\x02\x00\x00\x00\xc0'
>>> if data[:8] == valid_png_header:
...     width, height = struct.unpack('>LL', data[16:24])
...     print('Valid PNG, width', width, 'height', height)
... else:
...     print('Not a valid PNG')
...
Valid PNG, width 154 height 141
```

Этот код делает следующее.

- Переменная `data` содержит первые 30 байт файла PNG. Для того чтобы разместить ее на странице, я объединил две байтовые строки с помощью операторов `+` и `\`.
- Переменная `valid_png_header` содержит восьмибайтовую последовательность, которая обозначает начало корректного PNG-файла.
- Значение переменной `width` извлекается из 16–20-го байтов, а переменной `height` — из байтов 21–24.

`>LL` — это строка формата, которая указывает функции `unpack()`, как интерпретировать входные последовательности байтов и преобразовать их в типы данных Python. Рассмотрим ее детальнее:

- символ `<` означает, что целые числа хранятся в формате *big-endian (обратный порядок байтов)*;
- каждый символ `L` определяет четырехбайтное целое число типа `unsigned long`.

Вы можете проверить значение каждого четырехбайтного набора непосредственно:

```
>>> data[16:20]
b'\x00\x00\x00\x9a'
>>> data[20:24]0x9a
b'\x00\x00\x00\x8d'
```

У целых чисел с обратным порядком байтов главный байт располагается слева. Поскольку значения ширины и длины меньше 255, они умещаются в последний байт каждой последовательности. Вы можете убедиться в том, что эти шестнадцатеричные значения соответствуют ожидаемым десятичным значениям:

```
>>> 0x9a
154
>>> 0x8d
141
```

Если вы хотите отправить их в противоположном направлении и преобразовать данные Python в байты, используйте функцию `pack()` модуля `struct`:

```
>>> import struct
>>> struct.pack('>L', 154)
b'\x00\x00\x00\x9a'
>>> struct.pack('>L', 141)
b'\x00\x00\x00\x8d'
```

В табл. 7.5 и 7.6 показаны спецификаторы формата для функций `pack()` и `unpack()`. Спецификаторы порядка байтов располагаются первыми в строке формата.

Таблица 7.5. Спецификаторы порядка байтов

Спецификатор	Порядок байтов
<	Прямой порядок
>	Обратный порядок

Таблица 7.6. Спецификаторы формата

Спецификатор	Описание	Количество байтов
x	Пропустить байт	1
b	Знаковый байт	1
B	Беззнаковый байт	1
h	Знаковое короткое целое число	2
H	Беззнаковое короткое целое число	2
i	Знаковое целое число	4
I	Беззнаковое целое число	4
l	Знаковое длинное целое число	4
L	Беззнаковое длинное целое число	4
Q	Беззнаковое очень длинное целое число	8
f	Число с плавающей точкой	4
d	Число с плавающей точкой двойной точности	8
p	Счетчик и символы	1 + count
s	Символы	count

Спецификаторы типа следуют за символом, указывающим порядок байтов. Перед любым спецификатором может следовать число, которое указывает количество; запись `5B` аналогична записи `BBBBB`.

Вы можете использовать префикс счетчика вместо конструкции `>LL`:

```
>>> struct.unpack('>2L', data[16:24])
(154, 141)
```

Мы использовали разбиение `data[16:24]`, чтобы получить непосредственно интересующие нас байты. Мы также могли добавить спецификатор `x`, чтобы пропустить неинтересные части:

```
>>> struct.unpack('>16x2L6x', data)
(154, 141)
```

Эта строка означает:

- использовать формат с обратным порядком байтов (>);
- пропустить 16 байт (16x);
- прочесть 8 байт — два беззнаковых длинных целых числа (2L);
- пропустить последние 6 байт (6x).

Другие инструменты для работы с бинарными данными

Некоторые сторонние пакеты с открытым исходным кодом часто предлагают следующие более декларативные способы определения и извлечения бинарных данных:

- bitstring (http://bit.ly/py-bitstring);
- construct (http://bit.ly/py-construct);
- hachoir (http://bit.ly/hachoir-pkg);
- binio (http://spika.net/py/binio/).

В приложении Г содержатся инструкции о том, как загрузить и установить внешние пакеты вроде этих. Для следующего примера вам нужно установить пакет construct. Вот все, что вам необходимо сделать:

```
$ pip install construct
```

Вот так можно извлечь измерения PNG из нашей строки байтов data с помощью пакета construct:

```
>>> from construct import Struct, Magic, UBInt32, Const, String
>>> # адаптировано из кода по адресу https://github.com/construct
>>> fmt = Struct('png',
...     Magic(b'\x89PNG\r\n\x1a\n'),
...     UBInt32('length'),
...     Const(String('type', 4), b'IHDR'),
...     UBInt32('width'),
...     UBInt32('height')
...     )
>>> data = b'\x89PNG\r\n\x1a\n\x00\x00\x00\rIHDR' + \
...     b'\x00\x00\x00\x9a\x00\x00\x00\x8d\x08\x02\x00\x00\x00\xc0'
>>> result = fmt.parse(data)
>>> print(result)
Container:
    length = 13
    type = b'IHDR'
    width = 154
    height = 141
>>> print(result.width, result.height)
154, 141
```

Преобразование байтов/строк с помощью функции binascii()

Стандартный модуль binascii содержит функции, которые позволяют вам конвертировать данные в бинарный вид и в различные представления строк: шестнадцатеричное (с основанием 16), с основанием 64, uuencoded и др. Например, в следующем сниппете выведем на экран восьмибайтовый заголовок PNG как последовательность шестнадцатеричных значений вместо смеси символов ASCII и управляющих последовательностей вида \x xx, которые Python использует для отображения *байтовых* переменных:

```
>>> import binascii
>>> valid_png_header = b'\x89PNG\r\n\x1a\n'
>>> print(binascii.hexlify(valid_png_header))
b'89504e470d0a1a0a'
```

В другую сторону это тоже работает:

```
>>> print(binascii.unhexlify(b'89504e470d0a1a0a'))
b'\x89PNG\r\n\x1a\n'
```

Битовые операторы

Python предоставляет целочисленные операторы, работающие на уровне битов, их аналоги имеются в языке C. В табл. 7.7 показаны они все, а также примеры их использования для целых чисел a (в десятичной системе счисления 5, в двоичной — 0b0101) и b (в десятичной системе счисления 1, в двоичной — 0b0001).

Таблица 7.7. Целочисленные операции для уровня битов

Оператор	Описание	Пример	Десятичный результат	Двоичный результат
&	Логическое И	a & b	1	0b0001
|	Логическое ИЛИ	a | b	5	0b0101
^	Исключающее ИЛИ	a ^ b	4	0b0100
~	Инверсия битов	~a	–6	Двоичное представление зависит от размера типа int
<<	Сдвиг влево	a << 1	10	0b1010
>>	Сдвиг вправо	a << 1	2	0b0010

Эти операторы похожи на операторы для работы с множествами, показанные в главе 3. Оператор & возвращает биты, которые одинаковы в обоих аргументах, а оператор | возвращает биты, которые установлены в обоих аргументах. Оператор ^ возвращает биты, которые установлены в одном или в другом аргументе, но не в них обоих. Оператор ~ обращает порядок байтов в одном аргументе, он также

изменяет знак, поскольку старший бит целого числа указывает на его знак (1 означает «минус») в *арифметике дополнительных кодов*, которая используется во всех современных компьютерах. Операторы << и >> просто смещают биты влево или вправо. Сдвиг влево на один бит аналогичен умножению на 2, а сдвиг вправо — делению на 2.

Упражнения

1. Создайте строку Unicode с именем `mystery` и присвойте ей значение `'\U0001f4a9'`. Выведите на экран значение строки `mystery`. Найдите имя Unicode для `mystery`.
2. Закодируйте строку `mystery`, в этот раз с использованием кодировки UTF-8, в переменную типа `bytes` с именем `pop_bytes`. Выведите на экран значение переменной `pop_bytes`.
3. Используя кодировку UTF-8, декодируйте переменную `pop_bytes` в строку `pop_string`. Выведите на экран значение переменной `pop_string`. Равно ли оно значению переменной `mystery`?
4. Запишите следующее стихотворение с помощью старого стиля форматирования. Подставьте строки `'roast beef'`, `'ham'`, `'head'` и `'clam'` в эту строку:

```
My kitty cat likes %s,
My kitty cat likes %s,
My kitty cat fell on his %s
And now thinks he's a %s.
```

5. Запишите следующее письмо по форме с помощью форматирования нового стиля. Сохраните строку под именем `letter` (это имя вы используете в следующем упражнении):

```
Dear {salutation} {name},
Thank you for your letter. We are sorry that our {product} {verbed} in your
{room}. Please note that it should never be used in a {room}, especially
near any {animals}.
Send us your receipt and {amount} for shipping and handling. We will send
you another {product} that, in our tests, is {percent}% less likely to
have {verbed}.
Thank you for your support.
Sincerely,
{spokesman}
{job_title}
```

6. Создайте словарь с именем `response`, имеющий значения для строковых ключей `'salutation'`, `'name'`, `'product'`, `'verbed'` (прошедшее время от глагола verb), `'room'`, `'animals'`, `'amount'`, `'percent'`, `'spokesman'` и `'job_title'`. Выведите на экран значение переменной `letter`, в которую подставлены значения из словаря `response`.

7. При работе с текстом вам могут пригодиться регулярные выражения. Мы воспользуемся ими несколькими способами в следующем примере текста. Перед вами стихотворение *Ode on the Mammoth Cheese*, написанное Джеймсом Макинтайром (James McIntyre) в 1866 году во славу головки сыра весом 7000 фунтов, которая была сделана в Онтарио и отправлена в международное путешествие. Если не хотите вводить это стихотворение целиком, используйте свой любимый поисковик и скопируйте его текст в программу. Или скопируйте его из проекта «Гутенберг» (http://bit.ly/mcintyre-poetry). Назовите следующую строку `mammoth`:

```
We have seen thee, queen of cheese,
Lying quietly at your ease,
Gently fanned by evening breeze,
Thy fair form no flies dare seize.
All gaily dressed soon you'll go
To the great Provincial show,
To be admired by many a beau
In the city of Toronto.
Cows numerous as a swarm of bees,
Or as the leaves upon the trees,
It did require to make thee please,
And stand unrivalled, queen of cheese.
May you not receive a scar as
We have heard that Mr. Harris
Intends to send you off as far as
The great world's show at Paris.
Of the youth beware of these,
For some of them might rudely squeeze
And bite your cheek, then songs or glees
We could not sing, oh! queen of cheese.
We'rt thou suspended from balloon,
You'd cast a shade even at noon,
Folks would think it was the moon
About to fall and crush them soon.
```

8. Импортируйте модуль `re`, чтобы использовать функции регулярных выражений в Python. Используйте функцию `re.findall()`, чтобы вывести на экран все слова, которые начинаются с буквы «c».
9. Найдите все четырехбуквенные слова, которые начинаются с буквы «c».
10. Найдите все слова, которые заканчиваются на букву «r».
11. Найдите все слова, которые содержат три гласные подряд.
12. Используйте метод `unhexlify` для того, чтобы преобразовать шестнадцатеричную строку, созданную путем объединения двух строк, что позволило ей разместиться на странице, в переменную типа `bytes` с именем `gif`:

```
'47494638396101000100800000000000ffffff21f9' +
'0401000000002c000000000100010000020144003b'
```

13. Байты, содержащиеся в переменной `gif`, определяют однопиксельный прозрачный GIF-файл. Этот формат является одним из самых распространенных. Корректный файл формата GIF начинается со строки `GIF89a`. Является ли этот файл корректным?
14. Ширина файла формата GIF является шестнадцатибитным целым числом с обратным порядком байтов, которое начинается со смещения 6 байт. Его высота имеет такой же размер и начинается со смещения 8 байт. Извлеките и выведите на экран эти значения для переменной `gif`. Равны ли они 1?

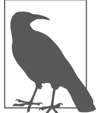
8 Данные должны куда-то попадать

Огромная ошибка — делать выводы, не имея необходимой информации.

Артур Конан Дойль

Активная программа работает с данными, которые хранятся в запоминающем устройстве с произвольным доступом (Random Access Memory (RAM)). RAM — очень быстрая память, но она дорога и требует постоянного питания; если питание пропадет, то все данные, которые в ней хранятся, будут утеряны. Жесткие диски медленнее оперативной памяти, но они более емкие, стоят дешевле и могут хранить данные даже после того, как кто-то выдернет шнур питания. Поэтому много усилий при создании компьютерных систем направлено на поиск лучшего соотношения между хранением данных на диске и в оперативной памяти. Как программистам, нам нужна *стойкость*: хранение и получение данных с помощью энергонезависимых медиа вроде дисков.

Эта глава посвящена разнообразным способам хранения данных, каждый из которых оптимизирован для разных целей: плоским файлам, структурированным файлам и базам данных. Операции с файлами, не касающиеся ввода-вывода, рассматриваются в разделе «Файлы» главы 10.



В этой главе также будут показаны первые примеры использования нестандартных модулей Python — да-да, этот код не входит в стандартные библиотеки Python. Вы можете без особых проблем установить их с помощью команды pip. Более подробно об использовании этих модулей вы можете прочитать в приложении Г.

Ввод информации в файлы и ее вывод из них

Самый простой пример стойкого хранилища — это старый добрый файл, иногда его еще называют *плоским файлом*. Он представляет собой последовательность байтов, которая хранится под *именем файла*. Вы *считываете* данные из файла

в память и *записываете* данные из памяти в файл. Python позволяет делать это довольно легко. Операции с файлами, присутствующие в этом языке программирования, были смоделированы на основе знакомых и популярных аналогов, имеющихся в Unix.

Перед тем как что-то записать в файл или считать из него, вам нужно открыть его:

```
fileobj = open(filename, mode)
```

Кратко поясню фрагменты этого вызова:

- `fileobj` — это объект файла, возвращаемый функцией `open()`;
- `filename` — это строка, представляющая собой имя файла;
- `mode` — это строка, указывающая на тип файла и действия, которые вы хотите над ним произвести.

Первая буква строки `mode` указывает на *операцию*:

- `r` означает чтение;
- `w` означает запись. Если файла не существует, он будет создан. Если файл существует, он будет перезаписан;
- `x` означает запись, но только если файла еще не существует;
- `a` означает добавление данных в конец файла, если он существует.

Вторая буква строки `mode` указывает на *тип* файла:

- `t` (или ничего) означает, что файл текстовый;
- `b` означает, что файл бинарный.

После открытия файла вы вызываете функции для чтения или записи данных, они будут показаны в следующих примерах.

Наконец, вам нужно *закрыть* файл.

Создадим файл, содержащий одну строку, в одной программе и считаем его в другой.

Запись в текстовый файл с помощью функции write()

По какой-то причине существует не так уж много лимериков о специальной теории относительности. В качестве источника данных придется использовать всего один:

```
>>> poem = '''There was a young lady named Bright,
... Whose speed was far faster than light;
... She started one day
... In a relative way,
... And returned on the previous night.'''
>>> len(poem)
150
```

Следующий код записывает это стихотворение в файл 'relativity' с помощью всего одного вызова:

```
>>> fout = open('relativity', 'wt')
>>> fout.write(poem)
150
>>> fout.close()
```

Функция write() возвращает число записанных байтов. Она не добавляет никаких пробелов или символов новой строки, как это делает функция print(). С помощью функции print() вы также можете записывать данные в текстовый файл:

```
>>> fout = open('relativity', 'wt')
>>> print(poem, file=fout)
>>> fout.close()
```

Отсюда возникает вопрос: какую функцию использовать — write() или print()? По умолчанию функция print() добавляет пробел после каждого аргумента и символ новой строки в конце. В предыдущем примере она добавила символ новой строки в файл relativity. Для того чтобы функция print() работала как функция write(), передайте ей два следующих аргумента:

- sep (разделитель, по умолчанию это пробел, ' ');
- end (символ конца файла, по умолчанию это символ новой строки, '\n').

Функция print() использует значения по умолчанию, если только вы не передадите ей что-то еще. Мы передадим ей пустые строки, чтобы подавить все лишние детали, обычно добавляемые функцией print():

```
>>> fout = open('relativity', 'wt')
>>> print(poem, file=fout, sep='', end='')
>>> fout.close()
```

Если исходная строка большая, вы можете записывать в файл ее фрагменты до тех пор, пока не запишете ее всю:

```
>>> fout = open('relativity', 'wt')
>>> size = len(poem)
>>> offset = 0
>>> chunk = 100
>>> while True:
...     if offset > size:
...         break
...     fout.write(poem[offset:offset+chunk])
...     offset += chunk
...
100
50
>>> fout.close()
```

Этот код записал 100 символов за первую попытку и последние 50 символов — за следующую.

Если файл relativity нам очень дорог, проверим, спасет ли режим x от его перезаписывания:

```
>>> fout = open('relativity', 'xt')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
FileExistsError: [Errno 17] File exists: 'relativity'
```

Вы можете использовать этот код вместе с обработчиком исключений:

```
>>> try:
...     fout = open('relativity', 'xt')]
...     fout.write('stomp stomp stomp')
... except FileExistsError:
...     print('relativity already exists!. That was a close one.')
...
relativity already exists!. That was a close one.
```

Считываем данные из текстового файла с помощью функций read(), readline() и readlines()

Вы можете вызвать функцию read() без аргументов, чтобы проглотить весь файл целиком, как показано в следующем примере. Будьте осторожны, делая это с крупными файлами, файл размером 1 Гбайт потребит 1 Гбайт памяти:

```
>>> fin = open('relativity', 'rt' )
>>> poem = fin.read()
>>> fin.close()
>>> len(poem)
150
```

Вы можете указать максимальное количество символов, которое функция read() вернет за один вызов. Давайте считывать по 100 символов за раз и присоединять каждый фрагмент к строке poem, чтобы воссоздать оригинал:

```
>>> poem = ''
>>> fin = open('relativity', 'rt' )
>>> chunk = 100
>>> while True:
...     fragment = fin.read(chunk)
...     if not fragment:
...         break
...     poem += fragment
...
>>> fin.close()
>>> len(poem)
150
```

После того как вы считали весь файл, дальнейшие вызовы функции `read()` будут возвращать пустую строку (`''`), которая будет оценена как `False` в проверке `if not fragment`. Это позволит выйти из цикла `while True`.

Вы также можете считывать файл по одной строке за раз с помощью функции `readline()`. В следующем примере мы будем присоединять каждую строку к строке `poem`, чтобы воссоздать оригинал:

```
>>> poem = ''
>>> fin = open('relativity', 'rt' )
>>> while True:
...     line = fin.readline()
...     if not line:
...         break
...     poem += line
...
>>> fin.close()
>>> len(poem)
150
```

Для текстового файла даже пустая строка имеет длину, равную `1` (символ новой строки), такая строка будет считаться `True`. Когда весь файл будет считан, функция `readline()` (как и функция `read()`) возвратит пустую строку, которая будет считаться `False`.

Самый простой способ считать текстовый файл — использовать *итератор*. Он будет возвращать по одной строке за раз. Этот пример похож на предыдущий, но кода в нем меньше:

```
>>> poem = ''
>>> fin = open('relativity', 'rt' )
>>> for line in fin:
...     poem += line
...
>>> fin.close()
>>> len(poem)
150
```

Во всех предыдущих примерах в результате получалась одна строка `poem`. Функция `readline()` считывает по одной строке за раз и возвращает список этих строк:

```
>>> fin = open('relativity', 'rt' )
>>> lines = fin.readlines()
>>> fin.close()
>>> print(len(lines), 'lines read')
5 lines read
>>> for line in lines:
...     print(line, end='')
...
```

```
There was a young lady named Bright,
Whose speed was far faster than light;
She started one day
In a relative way,
And returned on the previous night.>>>
```

Мы указали функции `print()` не добавлять автоматически символы новой строки, поскольку первые четыре строки сами их имеют. В последней строке этого символа не было, что заставило интерактивное приглашение появиться сразу после последней строки.

Записываем данные в бинарный файл с помощью функции write()

Если вы включите символ `'b'` в строку *режима*, файл будет открыт в бинарном режиме. В этом случае вы вместо чтения и записи строк будете работать с байтами.

У нас под рукой нет бинарного стихотворения, поэтому мы просто сгенерируем 256 байтовых значений от 0 до 255:

```
>>> bdata = bytes(range(0, 256))
>>> len(bdata)
256
```

Откроем файл для записи в бинарном режиме и запишем все данные сразу:

```
>>> fout = open('bfile', 'wb')
>>> fout.write(bdata)
256
>>> fout.close()
```

И вновь функция `write()` возвращает количество записанных байтов.

Как и в случае с текстом, вы можете записывать бинарные данные фрагментами:

```
>>> fout = open('bfile', 'wb')
>>> size = len(bdata)
>>> offset = 0
>>> chunk = 100
>>> while True:
...     if offset > size:
...         break
...     fout.write(bdata[offset:offset+chunk])
...     offset += chunk
...
100
100
56
>>> fout.close()
```

Читаем бинарные файлы с помощью функции read()

Это просто: все, что вам нужно, — открыть файл в режиме `'rb'`:

```
>>> fin = open('bfile', 'rb')
>>> bdata = fin.read()
>>> len(bdata)
256
>>> fin.close()
```

Закрываем файлы автоматически с помощью ключевого слова with

Если вы забудете закрыть за собой файл, его закроет Python после того, как будет удалена последняя ссылка на него. Это означает, что, если вы откроете файл и не закроете его явно, он будет закрыт автоматически по завершении функции. Но вы можете открыть файл внутри длинной функции или даже основного раздела программы. Файл должен быть закрыт, чтобы все оставшиеся операции записи были завершены.

У Python имеются *менеджеры контекста* для очистки объектов вроде открытых файлов. Вы можете использовать конструкцию `with` *выражение* `as` *переменная*:

```
>>> with open('relativity', 'wt') as fout:
...     fout.write(poem)
...
```

Вот и все. После того как блок кода, расположенный под менеджером контекста (в этом случае одна строка), завершится (или нормально, или путем генерации исключения), файл будет закрыт автоматически.

Меняем позицию с помощью функции seek()

По мере чтения и записи Python отслеживает ваше местоположение в файле. Функция `tell()` возвращает ваше текущее смещение от начала файла в байтах. Функция `seek()` позволяет вам перейти к другому смещению в файле. Это значит, что вам не обязательно читать каждый байт файла, чтобы добраться до последнего, — вы можете использовать функцию `seek()`, чтобы сместиться к последнему байту и считать его.

Для примера воспользуемся 256-байтным бинарным файлом `'bfile'`, который мы создали ранее:

```
>>> fin = open('bfile', 'rb')
>>> fin.tell()
0
```

Используем функцию `seek()`, чтобы перейти к предпоследнему байту файла:

```
>>> fin.seek(255)
255
```

Считаем все данные от текущей позиции до конца файла:

```
>>> bdata = fin.read()
>>> len(bdata)
1
>>> bdata[0]
255
```

Функция `seek()` также возвращает текущее смещение.

Вы также можете вызвать функцию `seek()`, передав ей второй аргумент: `seek(`*`offset`*`, `*`origin`*`)`:

- если значение `origin` равно `0` (по умолчанию), сместиться на *`offset`* байт с начала файла;
- если значение `origin` равно `1`, сместиться на *`offset`* байт с текущей позиции;
- если значение `origin` равно `2`, сместиться на *`offset`* байт с конца файла.

Эти значения также определены в стандартном модуле `os`:

```
>>> import os
>>> os.SEEK_SET
0
>>> os.SEEK_CUR
1
>>> os.SEEK_END
2
```

Благодаря этому мы можем считать последний байт разными способами:

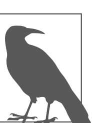
```
>>> fin = open('bfile', 'rb')
```

Один байт перед концом файла:

```
>>> fin.seek(-1, 2)
255
>>> fin.tell()
255
```

Считать данные до конца файла:

```
>>> bdata = fin.read()
>>> len(bdata)
1
>>> bdata[0]
255
```



Вам не нужно вызывать функцию tell(), чтобы работала функция seek(). Я только хотел показать, что обе эти функции возвращают одинаковое смещение.

Рассмотрим случай, когда мы вызываем функцию `seek()`, чтобы сместиться с текущей позиции:

```
>>> fin = open('bfile', 'rb')
```

Следующий пример переносит позицию за 2 байта до конца файла:

```
>>> fin.seek(254, 0)
254
>>> fin.tell()
254
```

Теперь перейдем вперед на 1 байт:

```
>>> fin.seek(1, 1)
255
>>> fin.tell()
255
```

Наконец, считаем все данные до конца файла:

```
>>> bdata = fin.read()
>>> len(bdata)
1
>>> bdata[0]
255
```

Эти функции наиболее полезны при работе с бинарными файлами. Вы можете использовать их и для работы с текстовыми файлами, но если файл содержит в себе не только символы формата ASCII (занимающие по одному байту в памяти), вам будет трудно определить смещение. Оно будет зависеть от кодировки текста, самая популярная кодировка (UTF-8) использует разное количество байтов для разных символов.

Структурированные текстовые файлы

Для простых текстовых файлов единственным уровнем организации является строка. Но иногда вам может понадобиться более структурированный файл. Вы можете захотеть сохранить данные своей программы для дальнейшего использования или отправить их другой программе.

Существует множество форматов, которые можно различить по следующим особенностям.

- *Разделитель*, символ вроде табуляции (`'\t'`), запятой (`','`) или вертикальной черточки (`'|'`). Это пример формата со значениями, разделенными запятой, (CSV).
- Символы `'<'` и `'>'`, окружающие *теги*. Примеры включают в себя XML и HTML.
- Знаки препинания. Примером является JavaScript Object Notation (JSON).

- Выделение пробелами. Примером является YAML (что в зависимости от источника может означать YAML Ain't Markup Language —«Не язык разметки», вам придется исследовать его самостоятельно).
- Прочие файлы, например конфигурационные.

Каждый из этих форматов структурированных файлов может быть считан и записан с помощью как минимум одного модуля Python.

CSV

Файлы с разделителями часто используются в качестве формата обмена данными для электронных таблиц и баз данных. Вы можете считать файл CSV вручную, по одной строке за раз, разделяя каждую строку на поля, расставляя запятые и добавляя результат в структуру данных вроде списка или словаря. Но лучшим решением будет использовать стандартный модуль `csv`, поскольку парсинг этих файлов может оказаться сложнее, чем вы думаете.

- Некоторые файлы имеют альтернативные разделители вместо запятой: самыми популярными являются `'|'` и `'\t'`.
- Некоторые файлы имеют *управляющие последовательности*. Если символ-разделитель встречается внутри поля, все поле может быть окружено символами кавычек или же перед ним будет находиться управляющая последовательность.
- Файлы имеют разные символы конца строк. В Unix используется `'\n'`, в Microsoft — `'\r \n'`, а Apple раньше применяла символ `'\r'`, но теперь перешла на использование `'\n'`.
- В первой строке могут содержаться названия колонок.

Для начала взглянем, как читать и записывать список строк, каждая из которых содержит список колонок:

```
>>> import csv
>>> villains = [
...     ['Doctor', 'No'],
...     ['Rosa', 'Klebb'],
...     ['Mister', 'Big'],
...     ['Auric', 'Goldfinger'],
     ['Ernst', 'Blofeld'],
...     ]
>>> with open('villains', 'wt') as fout:  # менеджер контекста
...     csvout = csv.writer(fout)
...     csvout.writerows(villains)
```

Этот код создает пять записей:

```
Doctor,No
Rosa,Klebb
Mister,Big
Auric,Goldfinger
Ernst,Blofeld
```

Теперь попробуем считать их обратно:

```
>>> import csv
>>> with open('villains', 'rt') as fin:  # менеджер контекста
...     cin = csv.reader(fin)
...     villains = [row for row in cin]  # Здесь используется включение списка
...
>>> print(villains)
[['Doctor', 'No'], ['Rosa', 'Klebb'], ['Mister', 'Big'],
['Auric', 'Goldfinger'], ['Ernst', 'Blofeld']]
```

Подумайте немного о включениях списка (в любой момент вы можете обратиться к разделу «Включения» главы 4, чтобы вспомнить синтаксис). Мы воспользовались структурой, созданной функцией `reader()`. Она услужливо создала в объекте `cin` ряды, которые мы можем извлечь с помощью цикла `for`.

Используя функции `reader()` и `writer()` с их стандартными опциями, мы получим колонки, которые разделены запятыми, и ряды, разделенные символами перевода строки.

Данные могут иметь формат списка словарей, а не списка списков. Снова считаем файл `villains`, в этот раз используя новую функцию `DictReader()` и указывая имена колонок:

```
>>> import csv
>>> with open('villains', 'rt') as fin:
...     cin = csv.DictReader(fin, fieldnames=['first', 'last'])
...     villains = [row for row in cin]
...
>>> print(villains)
[{'last': 'No', 'first': 'Doctor'},
{'last': 'Klebb', 'first': 'Rosa'},
{'last': 'Big', 'first': 'Mister'},
{'last': 'Goldfinger', 'first': 'Auric'},
{'last': 'Blofeld', 'first': 'Ernst'}]
```

Перепишем CSV-файл с помощью новой функции `DictWriter()`. Мы также вызовем функцию `writeheader()`, чтобы записать начальную строку, содержащую имена колонок, в CSV-файл:

```
import csv
villains = [
    {'first': 'Doctor', 'last': 'No'},
    {'first': 'Rosa', 'last': 'Klebb'},
    {'first': 'Mister', 'last': 'Big'},
    {'first': 'Auric', 'last': 'Goldfinger'},
    {'first': 'Ernst', 'last': 'Blofeld'},
    ]
with open('villains', 'wt') as fout:
    cout = csv.DictWriter(fout, ['first', 'last'])
    cout.writeheader()
    cout.writerows(villains)
```

Этот код создает файл villains со строкой заголовка:

```
first,last
Doctor,No
Rosa,Klebb
Mister,Big
Auric,Goldfinger
Ernst,Blofeld
```

Теперь считаем его обратно. Опуская аргумент fieldnames в вызове DictReader(), мы указываем функции использовать значения первой строки файла (first, last) как имена колонок и соответствующие ключи словаря:

```
>>> import csv
>>> with open('villains', 'rt') as fin:
...     cin = csv.DictReader(fin)
...     villains = [row for row in cin]
...
>>> print(villains)
[{'last': 'No', 'first': 'Doctor'},
{'last': 'Klebb', 'first': 'Rosa'},
{'last': 'Big', 'first': 'Mister'},
{'last': 'Goldfinger', 'first': 'Auric'},
{'last': 'Blofeld', 'first': 'Ernst'}]
```

XML

Файлы с разделителями охватывают только два измерения: ряды (строки) и колонки (поля внутри строк). Если вы хотите обмениваться структурами данных между программами, вам нужен способ кодировать иерархии, последовательности, множества и другие структуры с помощью текста.

XML является самым известным форматом *разметки*, который можно применять в этом случае. Для разделения данных он использует *теги*, как показано в следующем примере (файл menu.xml):

```
<?xml version="1.0"?>
<menu>
  <breakfast hours="7-11">
    <item price="$6.00">breakfast burritos</item>
    <item price="$4.00">pancakes</item>
  </breakfast>
  <lunch hours="11-3">
    <item price="$5.00">hamburger</item>
  </lunch>
  <dinner hours="3-10">
    <item price="8.00">spaghetti</item>
  </dinner>
</menu>
```

Рассмотрим основные характеристики формата XML.

- Теги начинаются с символа <. В этом примере использованы теги menu, breakfast, lunch, dinner и item.
- Пробелы игнорируются.
- Обычно после *начального тега* вроде <menu> следует остальной контент, а затем соответствующий *конечный тег* вроде </menu>.
- Теги могут быть *вложены* в другие теги на любой глубине. В этом примере теги item являются потомками тегов breakfast, lunch и dinner, которые, в свою очередь, являются потомками тега menu.
- Внутри стартового тега могут встретиться опциональные *атрибуты*. В этом примере price является опциональным атрибутом тега item.
- Теги могут содержать *значения*. В этом примере каждый тег item имеет значение вроде pancakes у второго элемента тега breakfast.
- Если у тега с именем thing нет значений или потомков, он может быть оформлен как единственный тег путем включения прямого слеша прямо перед закрывающей угловой скобкой (<thing/>), вместо того чтобы использовать начальный и конечный теги <thing> и </thing>.
- Место размещения данных — атрибуты, значения или теги-потомки — является в какой-то мере произвольным. Например, мы могли бы написать последний тег item как <item price="$8.00" food="spaghetti"/>.

XML часто используется в *каналах данных* и *сообщениях* и имеет подформаты вроде RSS и Atom. В некоторых отраслях, например в области бизнеса, имеются специализированные форматы XML (http://bit.ly/xml-finance).

Сверхгибкость формата XML вдохновила многих людей на создание библиотек для Python, каждая из которых отличается от других подходом и возможностями.

Самый простой способ проанализировать XML в Python — использовать библиотеку ElementTree. Рассмотрим небольшую программу, которая анализирует файл menu.xml и выводит на экран некоторые теги и атрибуты:

```
>>> import xml.etree.ElementTree as et
>>> tree = et.ElementTree(file='menu.xml')
>>> root = tree.getroot()
>>> root.tag
'menu'
>>> for child in root:
...     print('tag:', child.tag, 'attributes:', child.attrib)
     for grandchild in child:
...         print('\ttag:', grandchild.tag, 'attributes:', grandchild.attrib)
...
tag: breakfast attributes: {'hours': '7-11'}
    tag: item attributes: {'price': '$6.00'}
    tag: item attributes: {'price': '$4.00'}
```

```
tag: lunch attributes: {'hours': '11-3'}
    tag: item attributes: {'price': '$5.00'}
tag: dinner attributes: {'hours': '3-10'}
    tag: item attributes: {'price': '8.00'}
>>> len(root)     # количество разделов menu
3
>>> len(root[0])  # количество элементов breakfast
2
```

Для каждого элемента вложенных списков `tag` — это строка тега, а `attrib` — это словарь его атрибутов. Библиотека ElementTree имеет множество других способов поиска данных, организованных в формате XML, модификации этих данных и даже записи XML-файлов. Все детали изложены в документации библиотеки ElementTree (http://bit.ly/elementtree).

Среди других библиотек для работы с XML в Python можно отметить следующие:

- **xml.dom.** The Document Object Model (DOM), знакомая разработчикам на JavaScript, представляет веб-документы как иерархические структуры. Этот модуль загружает XML-файл в память целиком и позволяет вам получать доступ ко всем его частям;
- **xml.sax.** Simple API for XML, или SAX, разбирает XML на ходу, поэтому он не загружает в память сразу весь документ. Он может стать хорошим выбором, если вам нужно обработать очень большие потоки XML.

HTML

Огромные объемы данных сохраняются в формате гипертекстового языка разметки (Hypertext Markup Language, HTML), это основной формат документов в сети Интернет. Проблема заключается в том, что значительная часть этих документов не соответствует правилам формата HTML, поэтому его трудно разобрать. Помимо этого, большая часть HTML предназначена для того, чтобы форматировать выводимую информацию, а не обмениваться данными. Поскольку эта глава предназначена для того, чтобы описать относительно хорошо определенные форматы данных, я вынес рассмотрение HTML в главу 9.

JSON

JavaScript Object Notation (JSON) (http://www.json.org/) стал очень популярным форматом обмена данными, вышедшим за пределы языка JavaScript. Формат JSON является частью языка JavaScript и часто содержит легальный с точки зрения Python синтаксис. Он хорошо подходит Python, что делает его хорошим выбором при определении формата данных для обмена между программами. Вы увидите множество примеров использования JSON при веб-разработке в главе 9.

В отличие от XML, для которого написано множество модулей, для JSON существует всего один модуль с простым именем json. Эта программа кодирует (выгружает) данные в строку JSON и декодирует (загружает) строку JSON обратно. В следующем примере мы создадим структуру данных, содержащую данные из предыдущего примера, где описывался формат XML:

```
>>> menu = \
... {
... "breakfast": {
...         "hours": "7-11",
...         "items": {
...                 "breakfast burritos": "$6.00",
...                 "pancakes": "$4.00"
...                 }
...         },
... "lunch" : {
...         "hours": "11-3",
...         "items": {
...                 "hamburger": "$5.00"
...                 }
...         },
... "dinner": {
...         "hours": "3-10",
...         "items": {
...                 "spaghetti": "$8.00"
...                 }
...         }
... }
.
```

Далее закодируем структуру данных (menu) в строку JSON (menu_json) с помощью функции dumps():

```
>>> import json
>>> menu_json = json.dumps(menu)
>>> menu_json
'{"dinner": {"items": {"spaghetti": "$8.00"}, "hours": "3-10"},
"lunch": {"items": {"hamburger": "$5.00"}, "hours": "11-3"},
"breakfast": {"items": {"breakfast burritos": "$6.00", "pancakes":
"$4.00"}, "hours": "7-11"}}'
```

А теперь превратим строку JSON menu_json обратно в структуру данных (menu2) с помощью функции loads():

```
>>> menu2 = json.loads(menu_json)
>>> menu2
{'breakfast': {'items': {'breakfast burritos': '$6.00', 'pancakes':
'$4.00'}, 'hours': '7-11'}, 'lunch': {'items': {'hamburger': '$5.00'},
'hours': '11-3'}, 'dinner': {'items': {'spaghetti': '$8.00'}, 'hours': '3-10'}}
```

menu и menu2 являются словарями с одинаковыми ключами и значениями. Как всегда, в случае обычных словарей порядок, в котором вы получаете ключи, различается.

Вы можете получить исключение, пытаясь закодировать или декодировать некоторые объекты, включая такие объекты, как datetime (этот вопрос детально рассматривается в разделе «Календари и часы» главы 10), как показано здесь:

```
>>> import datetime
>>> now = datetime.datetime.utcnow()
>>> now
datetime.datetime(2013, 2, 22, 3, 49, 27, 483336)
>>> json.dumps(now)
Traceback (most recent call last):
# ... (deleted stack trace to save trees)
TypeError: datetime.datetime(2013, 2, 22, 3, 49, 27, 483336) is not JSON serializable
>>>
```

Это может случиться, поскольку стандарт JSON не определяет типы даты или времени — он ожидает, что вы укажете ему, как с ними работать. Вы можете преобразовать формат datetime во что-то, что JSON понимает, вроде строки или значения времени epoch (его мы рассмотрим в главе 10):

```
>>> now_str = str(now)
>>> json.dumps(now_str)
'"2013-02-22 03:49:27.483336"'
>>> from time import mktime
>>> now_epoch = int(mktime(now.timetuple()))
>>> json.dumps(now_epoch)
'1361526567'
```

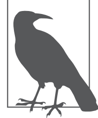
Если значение datetime встретится между нормальными сконвертированными типами данных, может быть неприятно выполнять такие особые преобразования. Вы можете изменить то, как JSON будет закодирован, с помощью наследования, что описано в разделе «Наследование» главы 6. Документация JSON для Python содержит пример такого переопределения для комплексных чисел, что также заставляет JSON притвориться мертвым. Напишем переопределение для datetime:

```
>>> class DTEncoder(json.JSONEncoder):
...     def default(self, obj):
...         # isinstance() checks the type of obj
...         if isinstance(obj, datetime.datetime):
...             return int(mktime(obj.timetuple()))
...         # else it's something the normal decoder knows:
...         return json.JSONEncoder.default(self, obj)
...
>>> json.dumps(now, cls=DTEncoder)
'1361526567'
```

Новый класс DTEncoder является подклассом, или классом-потомком, класса JSONEncoder. Нам нужно лишь переопределить его метод default(), добавив обработку datetime. Наследование гарантирует, что все остальное будет обработано родительским классом.

Функция isinstance() проверяет, является ли объект obj объектом класса datetime.datetime. Поскольку в Python все является объектом, функция isinstance() работает везде:

```
>>> type(now)
<class 'datetime.datetime'>
>>> isinstance(now, datetime.datetime)
True
>>> type(234)
<class 'int'>
>>> isinstance(234, int)
True
>>> type('hey')
<class 'str'>
>>> isinstance('hey', str)
True
```



Для JSON и других структурированных текстовых форматов вы можете загрузить файл в память и разместить его в структуре данных, не зная о самих структурах заранее. Далее вы можете с помощью функции isinstance() пройти по структурам и соответствующим типам методам, чтобы проверить их значения. Например, если один из элементов является словарем, вы можете извлечь его содержимое с помощью функций keys(), values() и items().

YAML

Как и JSON, YAML (http://www.yaml.org/) имеет ключи и значения, но обрабатывает большее количество типов данных, включая дату и время. Стандартная библиотека Python не содержит модулей, работающих с YAML, поэтому вам нужно установить стороннюю библиотеку yaml (http://pyyaml.org/wiki/PyYAML). Функция load() преобразует строку в формате YAML к данным Python, а функция dump() предназначена для противоположного действия.

Следующий YAML-файл, mcintyre.yaml, содержит информацию о канадском поэте Джеймсе Макинтайре (James McIntyre), в том числе два его стихотворения:

```
name:
  first: James
  last: McIntyre
dates:
  birth: 1828-05-25
```

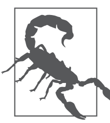
```
  death: 1906-03-31
details:
  bearded: true
  themes: [cheese, Canada]
books:
  url: http://www.gutenberg.org/files/36068/36068-h/36068-h.htm
poems:
  - title: 'Motto'
    text: |
      Politeness, perseverance and pluck,
      To their possessor will bring good luck.
  - title: 'Canadian Charms'
    text: |
      Here industry is not in vain,
      For we have bounteous crops of grain,
      And you behold on every field
      Of grass and roots abundant yield,
      But after all the greatest charm
      Is the snug home upon the farm,
      And stone walls now keep cattle warm.
```

Значения вроде `true`, `false`, `on` и `off` преобразуются в булевы переменные. Целые числа и строки преобразуются в их эквиваленты в Python. Для прочего синтаксиса создаются списки и словари:

```
>>> import yaml
>>> with open('mcintyre.yaml', 'rt') as fin:
>>>     text = fin.read()
>>> data = yaml.load(text)
>>> data['details']
{'themes': ['cheese', 'Canada'], 'bearded': True}
>>> len(data['poems'])
2
```

Создаваемые структуры данных совпадают со структурами YAML-файла, которые в данном случае имеют глубину более одного уровня. Вы можете получить заголовок второго стихотворения с помощью следующей ссылки:

```
>>> data['poems'][1]['title']
'Canadian Charms'
```



PyYAML может загружать объекты Python из строк, и это опасно. Используйте метод safe_load() вместо метода load(), если импортируете данные в формате YAML, которым не доверяете. А лучше всегда используйте метод safe_load(). Прочтите статью War is peace (http://nedbatchelder.com/blog/201302/war_is_peace.html), чтобы узнать о том, как незащищенная загрузка YAML скомпрометировала платформу Ruby on Rails.

Безопасность

Вы можете использовать любой формат, описанный в этой главе, для сохранения объектов в файлы и их считывания. Однако существует вероятность внедриться в этот процесс и вызвать проблемы с безопасностью.

Например, в следующем фрагменте XML-файла, состоящем из миллиарда усмешек, страница «Википедии» определяет десять вложенных сущностей, каждая из которых распространяется на более низкий уровень десять раз, порождая в сумме один миллиард сущностей:

```
<?xml version="1.0"?>
<!DOCTYPE lolz [
 <!ENTITY lol "lol">
 <!ENTITY lol1 "&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;">
 <!ENTITY lol2 "&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;">
 <!ENTITY lol3 "&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;">
 <!ENTITY lol4 "&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;">
 <!ENTITY lol5 "&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;">
 <!ENTITY lol6 "&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;">
 <!ENTITY lol7 "&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;">
 <!ENTITY lol8 "&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;">
 <!ENTITY lol9 "&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;">
]>
<lolz>&lol9;</lolz>
```

Плохая новость: миллиард усмешек подорвет работоспособность всех XML-библиотек, упомянутых в предыдущем разделе. На ресурсе Defused XML (https://bitbucket.org/tiran/defusedxml) эта и другие атаки перечислены наряду с уязвимостями библиотек Python. Перейдя по этой ссылке, вы увидите, как изменять настройки многих библиотек так, чтобы избежать этих проблем. Вы также можете использовать библиотеку defusedxml как внешний интерфейс безопасности для других библиотек:

```
>>> # insecure:
>>> from xml.etree.ElementTree import parse
>>> et = parse(xmlfile)
>>> # protected:
>>> from defusedxml.ElementTree import parse
>>> et = parse(xmlfile)
```

Конфигурационные файлы

Большинство программ предлагают различные *параметры* или *настройки*. Динамические настройки могут быть переданы как аргументы программы, но долговременные настройки должны где-то храниться. Искушение определить собственный формат *конфигурационного файла* быстро и неаккуратно очень сильно, но вы

должны устоять. Как правило, результат получаем неаккуратно, но не очень быстро. Вам нужно обслуживать как программу-писатель, так и программу-читатель (которая иногда называется *парсером*). Существуют хорошие альтернативы, которые вы можете добавить в свою программу, включая те, что были показаны в предыдущих разделах.

Здесь мы используем стандартный модуль `configparser`, который обрабатывает файлы с расширением `.ini`, характерные для Windows. Такие файлы имеют разделы, содержащие определения *`ключ = значение`*. Так выглядит минимальный файл `settings.cfg`:

```
[english]
greeting = Hello
[french]
greeting = Bonjour
[files]
home = /usr/local
# simple interpolation:
bin = %(home)s/bin
```

А так выглядит код, который позволяет считать его и разместить в структурах данных:

```
>>> import configparser
>>> cfg = configparser.ConfigParser()
>>> cfg.read('settings.cfg')
['settings.cfg']
>>> cfg
<configparser.ConfigParser object at 0x1006be4d0>
>>> cfg['french']
<Section: french>
>>> cfg['french']['greeting']
'Bonjour'
>>> cfg['files']['bin']
'/usr/local/bin'
```

Доступны и другие опции, включая более мощную интерполяцию. Обратитесь к документации `configparser` (http://bit.ly/configparser). Если вам нужно более двух уровней вложенности, попробуйте использовать YAML или JSON.

Другие форматы обмена данными

Такие бинарные форматы обмена данными, как MsgPack (http://msgpack.org/), Protocol Buffers (https://code.google.com/p/protobuf/), Avro (http://avro.apache.org/docs/current/), Thrift (http://thrift.apache.org/), обычно компактнее и быстрее, чем XML или JSON. Поскольку они бинарные, ни один из них не может быть изменен человеком, вооружившимся текстовым редактором.

Сериализация с помощью pickle

Сохранение структур данных в файл называется *сериализацией*. Форматы вроде JSON могут требовать наличия пользовательских преобразователей для сериализации всех типов данных программы, написанной на Python. Python предоставляет модуль pickle, позволяющий сохранить и восстановить любой объект в специальном бинарном формате.

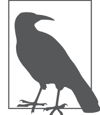
Помните, как JSON сошел с ума, когда встретил объект datetime? Для pickle это не проблема:

```
>>> import pickle
>>> import datetime
>>> now1 = datetime.datetime.utcnow()
>>> pickled = pickle.dumps(now1)
>>> now2 = pickle.loads(pickled)
>>> now1
datetime.datetime(2014, 6, 22, 23, 24, 19, 195722)
>>> now2
datetime.datetime(2014, 6, 22, 23, 24, 19, 195722)
```

pickle работает также с вашими собственными классами и объектами. Мы определим небольшой класс, который называется Tiny и возвращает слово 'tiny', когда он используется как строка:

```
>>> import pickle
>>> class Tiny():
...     def __str__(self):
...         return 'tiny'
...
>>> obj1 = Tiny()
>>> obj1
<__main__.Tiny object at 0x10076ed10>
>>> str(obj1)
'tiny'
>>> pickled = pickle.dumps(obj1)
>>> pickled
b'\x80\x03c__main__\nTiny\nq\x00)\x81q\x01.'
>>> obj2 = pickle.loads(pickled)
>>> obj2
<__main__.Tiny object at 0x10076e550>
>>> str(obj2)
'tiny'
```

pickled — это обработанная pickle бинарная строка, созданная из объекта obj1. Мы преобразовали ее в объект obj2, чтобы сделать копию объекта obj1. Используйте функцию dump(), чтобы pickle сохранил данные в файл, и функцию load(), чтобы pickle загрузил данные из файла.



Поскольку pickle может создавать объекты Python, к нему применимы предупреждения о безопасности, которые были рассмотрены ранее. Не загружайте в pickle данные, которым не доверяете.

Структурированные бинарные файлы

Некоторые файловые форматы были разработаны для того, чтобы хранить определенные структуры данных, и они не являются ни реляционными, ни базами данных NoSQL. В следующих разделах рассказывается о некоторых из них.

Электронные таблицы

Электронные таблицы, в частности Microsoft Excel, — это широко распространенный формат данных. Если вы можете сохранить свою таблицу в CSV-файл, то можете считать его с помощью стандартного модуля `csv`, который был описан ранее. Если у вас есть бинарный файл `xls`, для его считывания и записи можете использовать стороннюю библиотеку xlrd.

HDF5

HDF5 (http://www.hdfgroup.org/why_hdf) — это бинарный формат данных, предназначенный для хранения многомерных или иерархических числовых данных. Обычно он используется в научных целях, где быстрый случайный доступ к крупным наборам данных (от гигабайтов до терабайтов) является распространенным требованием. Несмотря на то что HDF5 в некоторых случаях мог бы стать хорошей альтернативой базам данных, по каким-то причинам этот формат практически неизвестен в современном мире. Он лучше всего подходит для приложений вида WORM (write once/read many — «запиши однажды — считай много раз»), которые не нуждаются в защите от конфликтующих записей. Вы можете счесть полезными следующие модули:

- h5py — является интерфейсом низкого уровня с широкими возможностями. Прочтите его документацию (http://www.h5py.org/) и код (https://github.com/h5py/h5py);
- PyTables — это интерфейс немного более высокого уровня, имеющий некоторые особенности, характерные для баз данных. Прочтите его документацию (http://www.pytables.org/) и код (http://pytables.github.com/).

Оба этих формата рассматриваются в приложении В с точки зрения применения в научных приложениях, написанных на Python. Здесь я упоминаю об HDF5 затем, чтобы у вас был под рукой нестандартный вариант на случай, когда вам нужно сохранять и вычитывать крупные объемы данных. Хорошим примером использования этого формата является Million Song Dataset (http://bit.ly/millionsong), содержащий информацию о песнях.

Реляционные базы данных

Реляционным базам данных всего около 40 лет, но в компьютерном мире они используются повсеместно. Вам практически наверняка придется поработать с ними. В эти моменты вы сможете оценить следующие их преимущества.

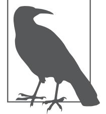
- Доступ к данным возможен для нескольких пользователей одновременно.
- Действует защита от повреждения данных пользователями.
- Существуют эффективные методы сохранения и считывания данных.
- Данные определяются *схемами*, их можно *ограничить*.
- *Объединения* позволяют найти отношения между различными типами данных.
- Декларативный (в противоположность императивному) язык запросов SQL (Structured Query Language, структурированный язык запросов).

Такие базы данных называются *реляционными*, поскольку они показывают отношения между различными типами данных, представленными в форме *таблиц* (в наши дни они называются именно так). Например, в нашем примере в меню существовало бы отношение между каждым элементом и его ценой.

Таблица представляет собой сетку с рядами и графами, похожую на электронную таблицу. Чтобы создать таблицу, необходимо указать ее имя и порядок, имена и типы ее граф. Каждый ряд имеет одинаковые графы, однако графа может быть определена так, что в ней можно ничего не размещать (`null`). В примере с меню вы могли бы создать таблицу, содержащую по одному ряду для каждого продаваемого элемента. Каждый элемент имеет одинаковые графы, включая ту, которая хранит цену.

Первичным ключом таблицы является графа или группа граф, их значения должны быть уникальными. Это предотвращает ввод одинаковых данных в таблицу. Этот ключ индексируется для более быстрого поиска по время выполнения запроса. Работа индекса немного похожа на алфавитный указатель, что позволяет быстро найти определенный ряд.

Каждая таблица находится внутри родительской *базы данных*, что напоминает файлы в каталоге. Два уровня иерархии позволяют немного лучше организовывать данные.



Да, словосочетание «база данных» используется в нескольких случаях: когда разговор идет о сервере, о хранилище таблиц и о данных, которые там хранятся. Если вам нужно упомянуть их одновременно, можно назвать их сервером базы данных, базой данных и данными.

Если вам нужно найти ряды по некоторому неключевому значению, определите для столбца *вторичный индекс*. В противном случае база данных должна будет выполнить *сканирование таблицы* — поиск нужного значения перебором всех рядов.

Таблицы могут быть связаны друг с другом с помощью *внешних ключей*, и значения граф могут быть ограничены этими ключами.

SQL

SQL не является API или протоколом. Это декларативный язык: вы говорите, что вам нужно, вместо того, как это сделать. Это универсальный язык реляционных баз данных. Запросы SQL являются текстовыми строками, которые клиент отсылает серверу базы данных, определяющему, что с ними делать дальше.

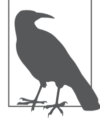
Существует несколько стандартов определения SQL, но все поставщики баз данных добавили свои модификации и расширения, что вылилось в возникновение множества *диалектов* SQL. Если вы храните данные в реляционной базе данных, SQL дает вам некоторую переносимость данных. Однако наличие диалектов и операционных различий может усложнить перенос данных в другую базу.

Существуют две основные категории утверждений SQL: DDL *(Data Definition Language, язык определения данных)*, который обрабатывает создание, удаление, ограничения и разрешения для таблиц, баз данных и использует DML *(Data Manipulation Language, язык манипулирования данными)*, который обрабатывает добавление данных, их выборку, обновление и удаление.

В табл. 8.1 перечислены основные команды SQL DDL.

Таблица 8.1. Основные команды SQL DDL

Операция	**Шаблон SQL**	**Пример SQL**
Создание базы данных	CREATE DATABASE dbname	CREATE DATABASE d
Выбор текущей базы данных	USE dbname	USE d
Удаление базы данных и ее таблиц	DROP DATABASE dbname	DROP DATABASE d
Создание таблицы	CREATE TABLE tbname (coldefs)	CREATE TABLE t (id INT, count INT)
Удаление таблицы	DROP TABLE tbname	DROP TABLE t
Удаление всех строк таблицы	TRUNCATE TABLE tbname	TRUNCATE TABLE t



Почему все пишется БОЛЬШИМИ БУКВАМИ? Язык SQL не зависит от регистра, но по традиции (не спрашивайте меня почему) ключевые слова ВЫКРИКИВАЮТСЯ, чтобы можно было отличить их от имен граф.

Основные операции DML реляционной базы данных можно запомнить с помощью акронима CRUD:

- Create — создание с помощью оператора SQL `INSERT`;
- Read — чтение с помощью `SELECT`;
- Update — обновление с помощью `UPDATE`;
- Delete — удаление с помощью `DELETE`.

В табл. 8.2 показаны команды, доступные SQL DML.

Таблица 8.2. Основные команды SQL DML

Операция	Шаблон SQL	Пример SQL
Добавление ряда	INSERT INTO tbname VALUES(…)	INSERT INTO t VALUES(7, 40)
Выборка всех рядов и граф	SELECT * FROM tbname	SELECT * FROM t
Выборка всех рядов и некоторых граф	SELECT cols FROM tbname	SELECT id, count FROM t
Выборка некоторых рядов и некоторых граф	SELECT cols FROM tbname WHERE condition	SELECT id, count from t WHERE count > 5 AND id = 9
Изменение некоторых рядов в графе	UPDATE tbname SET col = value WHERE condition	UPDATE t SET count = 3 WHERE id = 5
Удаление некоторых рядов	DELETE FROM tbname WHERE condition	DELETE FROM t WHERE count <= 10 OR id = 16

DB-API

Программный интерфейс приложения (Application Programming Interface, API) — это набор функций, которые вы можете вызвать, чтобы получить доступ к какой-либо услуге. DB-API (http://bit.ly/db-api) — это стандартный API в Python, предназначенный для получения доступа к реляционным базам данных. С его помощью вы можете написать одну программу, которая работает с несколькими видами реляционных баз данных, вместо того чтобы писать несколько программ для работы с каждым видом баз данных по отдельности. Этот API похож на JDBC в Java или dbi в Perl.

Рассмотрим его основные функции.

- `connect()` — создание соединения с базой данных. Этот вызов может включать в себя аргументы вроде имени пользователя, пароля, адреса сервера и пр.
- `cursor()` — создание объекта *курсора*, предназначенного для работы с запросами.
- `execute()` и `executemany()` — запуск одной или более команд SQL.
- `fetchone()`, `fetchmany()` и `fetchall()` — получение результатов работы функции `execute`.

Модули работы с базами данных в Python, которые будут рассмотрены в следующих разделах, соответствуют DB-API, часто имея некоторые расширения или разницу в деталях.

SQLite

SQLite (http://www.sqlite.org/) — это хорошая легковесная реляционная база данных с открытым исходным кодом. Она реализована как стандартная библиотека Python и хранит базы данных в обычных файлах. Эти файлы можно переносить

на другие машины и в операционные системы, что делает SQLite очень портативным решением для создания простых реляционных баз данных. У нее не так много возможностей, как у MySQL или PostgreSQL, но она поддерживает SQL и позволяет нескольким пользователям работать с ней одновременно. Браузеры, смартфоны и другие операционные системы используют SQLite как встроенную базу данных.

Работа с базой данных начинается с вызова `connect()` для установки соединения с локальным файлом базы данных, который вы хотите создать или использовать. Этот файл эквивалентен похожей на каталог *базе данных*, которая хранит таблицы на других серверах. С помощью специальной строки `':memory:'` можно создать базу данных только в памяти — это быстро и полезно для тестирования, но данные будут потеряны при завершении программы или выключении компьютера.

Для следующего примера создадим базу данных `enterprise.db` и таблицу `zoo`, чтобы управлять нашим увлекательным бизнесом по содержанию придорожного контактного зоопарка. В таблице будут содержаться следующие графы:

- `critter` — строка переменной длины, наш первичный ключ;
- `count` — целочисленное количество единиц используемого инвентаря для этого животного;
- `damages` — количество долларов, потерянных из-за взаимодействий людей с животными:

```
>>> import sqlite3
>>> conn = sqlite3.connect('enterprise.db')
>>> curs = conn.cursor()
>>> curs.execute('''CREATE TABLE zoo
    (critter VARCHAR(20) PRIMARY KEY,
     count INT,
     damages FLOAT)''')
<sqlite3.Cursor object at 0x1006a22d0>
```

Тройные кавычки в Python очень полезны при создании длинных строк вроде запросов SQL.

Теперь добавим в зоопарк несколько животных:

```
>>> curs.execute('INSERT INTO zoo VALUES("duck", 5, 0.0)')
<sqlite3.Cursor object at 0x1006a22d0>
>>> curs.execute('INSERT INTO zoo VALUES("bear", 2, 1000.0)')
<sqlite3.Cursor object at 0x1006a22d0>
```

Существует более безопасный способ добавить данные — использовать заполнитель:

```
>>> ins = 'INSERT INTO zoo (critter, count, damages) VALUES(?, ?, ?)'
>>> curs.execute(ins, ('weasel', 1, 2000.0))
<sqlite3.Cursor object at 0x1006a22d0>
```

В этот раз мы использовали в запросе три вопросительных знака, чтобы показать, что планируем добавить три значения, а затем добавить эти значения списком в функции execute(). Заполнители помогают нам справляться с нудными деталями вроде расстановки кавычек. Они защищают от *SQL-инъекций* — внешней атаки, распространенной в Сети, которая внедряет в систему вредные команды SQL.

Теперь проверим, сможем ли мы получить назад список наших животных:

```
>>> curs.execute('SELECT * FROM zoo')
<sqlite3.Cursor object at 0x1006a22d0>
>>> rows = curs.fetchall()
>>> print(rows)
[('duck', 5, 0.0), ('bear', 2, 1000.0), ('weasel', 1, 2000.0)]
```

Получим их снова, но на этот раз упорядочим список по количеству животных:

```
>>> curs.execute('SELECT * from zoo ORDER BY count')
<sqlite3.Cursor object at 0x1006a22d0>
>>> curs.fetchall()
[('weasel', 1, 2000.0), ('bear', 2, 1000.0), ('duck', 5, 0.0)]
```

Эй, мы хотели получить список в нисходящем порядке:

```
>>> curs.execute('SELECT * from zoo ORDER BY count DESC')
<sqlite3.Cursor object at 0x1006a22d0>
>>> curs.fetchall()
[('duck', 5, 0.0), ('bear', 2, 1000.0), ('weasel', 1, 2000.0)]
```

Какие животные обходятся нам дороже всего?

```
>>> curs.execute('''SELECT * FROM zoo WHERE
...     damages = (SELECT MAX(damages) FROM zoo)''')
<sqlite3.Cursor object at 0x1006a22d0>
>>> curs.fetchall()
[('weasel', 1, 2000.0)]
```

Вы могли бы подумать, что это медведи. Лучше всегда проверять актуальные данные.

Перед тем как оставить в покое SQLite, нам нужно прибраться. Если мы открывали соединение и курсор, нужно закрыть их после того, как работа будет закончена:

```
>>> curs.close()
>>> conn.close()
```

MySQL

MySQL (http://www.mysql.com/) — это очень популярная реляционная база данных с открытым исходным кодом. В отличие от SQLite она является настоящим сер-

вером, поэтому клиенты могут получать к ней доступ с разных устройств всей сети.

MysqlDB (http://sourceforge.net/projects/mysql-python) является самым популярным драйвером для MySQL, но его еще не портировали в Python 3. В табл. 8.3 перечислены драйверы, которые вы можете использовать для того, чтобы получить доступ к MySQL из Python.

Таблица 8.3. Драйверы MySQL

Название	**Ссылка**	**Пакет PyPi**	**Импортировать как**	**Примечание**
MySQL Connector	http://bit.ly/mysql-cpdg	mysql-connector-python	mysql.connector	—
PYMySQL	https://github.com/petehunt/PyMySQL/	pymysql	pymysql	—
oursql	http://pythonhosted.org/oursql/	oursql	oursql	Требует наличия клиентской библиотеки MySQL C

PostgreSQL

PostgreSQL (http://www.postgresql.org/) — реляционная база данных с открытым исходным кодом, имеющая широкие возможности и гораздо более продвинутая, чем MySQL. В табл. 8.4 показаны драйверы Python, которые вы можете использовать для того, чтобы получить к ней доступ.

Таблица 8.4. Драйверы PostgreSQL

Название	**Ссылка**	**Пакет PyPi**	**Импортировать как**	**Примечание**
psycopg2	http://initd.org/psycopg/	psycopg2	psycopg2	Необходим pg_config из клиентских инструментов PostgreSQL
py-postgresql	http://python.projects.pgfoundry.org/	py-postgresql	postgresql	—

SQLAlchemy

SQL не во всех реляционных базах данных одинаков, и DB-API дает вам ограниченный набор возможностей. Каждая база данных реализует определенный *диалект*, отражая свои особенности и философию. Многие библиотеки пытаются тем или иным способом компенсировать эти различия. Самая популярная библиотека для работы с разными базами данных — SQLAlchemy (http://www.sqlalchemy.org/).

Эта библиотека не является стандартной, но она широко известна и используется многими людьми. Вы можете установить ее в свою систему с помощью следующей команды:

```
$ pip install sqlalchemy
```

Можете использовать SQLAlchemy на нескольких уровнях.

- На самом низком уровне она работает с *пулами* соединений к базе данных, выполняет команды SQL и возвращает результат. Этот уровень очень похож на DB-API.
- Следующий уровень — *язык выражений* SQL, построитель SQL в Python.
- Самый высокий уровень — это слой ORM (Object Relational Model, объектно-реляционное отображение), который использует язык выражений SQL Expression Language и связывает код приложения с реляционными структурами данных.

По мере углубления в материал вы поймете, что означают эти термины. SQLAlchemy работает с драйверами базы данных, задокументированными в предыдущих разделах. Вам не нужно импортировать драйвер — он будет определен с помощью строки соединения, которую вы предоставляете SQLAlchemy. Эта строка выглядит примерно так:

```
dialect + driver :// user : password @ host : port / dbname
```

В эту строку нужно поместить следующие значения:

- `dialect` — тип базы данных;
- `driver` — драйвер, который вы хотите использовать для этой базы данных;
- `user` и `password` — строки аутентификации для этой базы данных;
- `host` и `port` — расположение сервера базы данных (значение `port` нужно указывать только в том случае, если вы используете нестандартный порт);
- `dbname` — имя базы данных, к которой нужно подключиться.

В табл. 8.5 перечислены диалекты и драйверы.

Таблица 8.5. Соединение с SQLAlchemy

Диалект	**Драйвер**
sqlite	pysqlite (можно опустить)
mysql	mysqlconnector
mysql	pymysql
mysql	oursql
postgresql	psycopg2
postgresql	pypostgresql

Уровень движка

Сначала мы попробуем поработать с самым низким уровнем SQLAlchemy, возможности которого почти не отличаются от функций DB-API.

Попробуем поработать с SQLite, поскольку его поддержка уже встроена в Python. Строка соединения для SQLite опускает значения параметров host, port, user и password. dbname информирует SQLite о том, какой файл использовать для хранения вашей базы данных. Если вы опустите параметр dbname, SQLite создаст базу данных в памяти. Если значение параметра dbname начинается со слеша (/), оно является абсолютным именем файла на вашем компьютере (как в Linux и OS X). В противном случае оно является относительным именем текущего каталога.

Следующие сегменты являются частью одной программы, разделенной на части для удобства объяснения.

Для начала нужно импортировать все, что нам понадобится. Следующая строка является примером *импортирования псевдонима*, который позволяет использовать строку sa для того, чтобы ссылаться на методы SQLAlchemy. Я делаю это в основном потому, что sa написать гораздо проще, чем sqlalchemy:

```
>>> import sqlalchemy as sa
```

Соединимся с базой данных и создадим хранилище в памяти (строка аргументов 'sqlite:///:memory: ' также сработает):

```
>>> conn = sa.create_engine('sqlite://')
```

Создадим таблицу, которая называется zoo и содержит три графы:

```
>>> conn.execute('''CREATE TABLE zoo
...     (critter VARCHAR(20) PRIMARY KEY,
...      count INT,
...      damages FLOAT)''')
<sqlalchemy.engine.result.ResultProxy object at 0x1017efb10>
```

Вызов conn.execute() возвращает объект SQLAlchemy, который называется ResultProxy. Скоро вы увидите, что с ним можно сделать.

Кстати, если вы раньше никогда не создавали базы данных, примите мои поздравления. Можете вычеркнуть этот пункт из своего списка дел, которые обязательно нужно реализовать в жизни.

Далее вставьте три набора данных в новую пустую таблицу:

```
>>> ins = 'INSERT INTO zoo (critter, count, damages) VALUES (?, ?, ?)'
>>> conn.execute(ins, 'duck', 10, 0.0)
<sqlalchemy.engine.result.ResultProxy object at 0x1017efb50>
>>> conn.execute(ins, 'bear', 2, 1000.0)
<sqlalchemy.engine.result.ResultProxy object at 0x1017ef090>
>>> conn.execute(ins, 'weasel', 1, 2000.0)
<sqlalchemy.engine.result.ResultProxy object at 0x1017ef450>
```

Далее сделайте выборку того, что только что разместили в базе:

```
>>> rows = conn.execute('SELECT * FROM zoo')
```

В SQLAlchemy `rows` не является списком — это специальный объект `ResultProxy`, который мы не можем отобразить непосредственно:

```
>>> print(rows)
<sqlalchemy.engine.result.ResultProxy object at 0x1017ef9d0>
```

Однако вы можете итерировать по нему, как по списку, и получать по одному ряду за раз:

```
>>> for row in rows:
...     print(row)
...
('duck', 10, 0.0)
('bear', 2, 1000.0)
('weasel', 1, 2000.0)
```

Этот пример очень похож на другой, где использовался SQLite DB-API. Единственное преимущество этого подхода заключается в том, что нам не нужно импортировать драйвер — SQLAlchemy сам определил драйвер на основе строки соединения. Простое изменение строки соединения позволит перенести этот код на базу данных другого типа. Еще один плюс SQLAlchemy заключается в наличии *пула соединений*, о котором вы можете прочитать на сайте http://bit.ly/conn-pooling, содержащем документацию.

Язык выражений SQL

Следующий уровень SQLAlchemy — это язык выражений SQL. Он предоставляет функции, которые позволяют создать SQL для разных операций. Язык выражений обрабатывает большее количество различий в диалектах, чем низкоуровневый слой движка. Он может оказаться полезным промежуточным решением для приложений, работающих с реляционными базами данных.

Рассмотрим создание и наполнение таблицы `zoo`. Вновь все последующие фрагменты принадлежат одной программе.

Импортирование и подключение не изменяются:

```
>>> import sqlalchemy as sa
>>> conn = sa.create_engine('sqlite://')
```

Для того чтобы определить таблицу `zoo`, вместо SQL начнем использовать язык выражений:

```
>>> meta = sa.MetaData()
>>> zoo = sa.Table('zoo', meta,
```

```
...     sa.Column('critter', sa.String, primary_key=True),
...     sa.Column('count', sa.Integer),
...     sa.Column('damages', sa.Float)
...    )
>>> meta.create_all(conn)
```

Обратите внимание на круглые скобки в операции, которая занимает несколько строк в предыдущем примере. Структура метода `Table()` совпадает со структурой таблицы. Поскольку наша таблица содержит три графы, в методе `Table()` расположены три вызова метода `Column()`.

`zoo` представляет собой некий волшебный объект, который соединяет мир баз данных SQL и мир структур данных Python.

Запишите в таблицу данные с помощью новых функций языка выражений:

```
... conn.execute(zoo.insert(('bear', 2, 1000.0)))
<sqlalchemy.engine.result.ResultProxy object at 0x1017ea910>
>>> conn.execute(zoo.insert(('weasel', 1, 2000.0)))
<sqlalchemy.engine.result.ResultProxy object at 0x1017eab10>
>>> conn.execute(zoo.insert(('duck', 10, 0)))
<sqlalchemy.engine.result.ResultProxy object at 0x1017eac50>
```

Далее создадим оператор `SELECT` (`zoo.select()` делает выборку всего, что содержится в таблице, представленной объектом `zoo`, как это сделала бы инструкция `SELECT * FROM zoo` в простом SQL):

```
>>> result = conn.execute(zoo.select())
```

Наконец, получим результат:

```
>>> rows = result.fetchall()
>>> print(rows)
[('bear', 2, 1000.0), ('weasel', 1, 2000.0), ('duck', 10, 0.0)]
```

The Object-Relational Mapper

В предыдущем разделе объект `zoo` являлся промежуточным звеном между SQL и Python. В самом верхнем слое SQLAlchemy объектно-реляционное отображение (Object-Relational Mapper, ORM) использует язык выражений SQL, но старается сделать реальные механизмы базы данных невидимыми. Вы определяете классы, а ORM обрабатывает способ, с помощью которого они получают данные из базы данных и возвращают их обратно. Основная идея, на которой базируется сложный термин «объектно-реляционное отображение», заключается в том, что вы можете ссылаться на объекты в своем коде и поэтому придерживаться принципов работы с Python, но при этом использовать реляционную базу данных.

Мы определим класс `Zoo` и свяжем его с ORM. В этот раз укажем SQLite использовать файл `zoo.db`, чтобы убедиться, что ORM работает.

Как и в предыдущих двух разделах, следующие сниппеты являются частью одной программы, разбитой на фрагменты, которые я объясню. Не переживайте, если чего-то не поймете. В документации к SQLAlchemy содержатся все необходимые детали, работа с SQLALchemy может оказаться довольно сложной. Я хочу показать вам, что нужно сделать, чтобы ORM работал, чтобы вы могли определить, какой из подходов, рассмотренных в этой главе, годится для вас больше других.

Импорт остается неизменным, но в этот раз нам нужно кое-что еще:

```
>>> import sqlalchemy as sa
>>> from sqlalchemy.ext.declarative import declarative_base
```

Вот так создается соединение:

```
>>> conn = sa.create_engine('sqlite:///zoo.db')
```

Теперь мы начинаем работать с SQLAlchemy ORM. Определяем класс `Zoo` и связываем его атрибуты с графами таблицы:

```
>>> Base = declarative_base()
>>> class Zoo(Base):
...     __tablename__ = 'zoo'
...     critter = sa.Column('critter', sa.String, primary_key=True)
...     count = sa.Column('count', sa.Integer)
...     damages = sa.Column('damages', sa.Float)
...     def __init__(self, critter, count, damages):
...         self.critter = critter
...         self.count = count
...         self.damages = damages
...     def __repr__(self):
...       return "<Zoo({}, {}, {})>".format(self.critter, self.count, self.damages)
```

Следующая строка как по волшебству создает базу данных и таблицу:

```
>>> Base.metadata.create_all(conn)
```

Вы можете добавить в таблицу данные путем создания объектов Python. ORM управляет данными изнутри:

```
>>> first = Zoo('duck', 10, 0.0)
>>> second = Zoo('bear', 2, 1000.0)
>>> third = Zoo('weasel', 1, 2000.0)
>>> first
<Zoo(duck, 10, 0.0)>
```

Далее мы указываем ORM отвезти нас в страну SQL. Создаем сессию, чтобы беседовать с базой данных:

```
>>> from sqlalchemy.orm import sessionmaker
>>> Session = sessionmaker(bind=conn)
>>> session = Session()
```

Внутри сессии записываем три созданных нами объекта в базу данных. Функция `add()` добавляет один объект, а функция `add_all()` добавляет список:

```
>>> session.add(first)
>>> session.add_all([second, third])
```

Наконец, нам нужно завершить сессию:

```
>>> session.commit()
```

Сработало? Файл `zoo.db` был создан в текущем каталоге. Вы можете использовать программу командной строки `sqlite3`, чтобы убедиться в этом:

```
$ sqlite3 zoo.db
SQLite version 3.6.12
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
sqlite> .tables
zoo
sqlite> select * from zoo;
duck|10|0.0
bear|2|1000.0
weasel|1|2000.0
```

Цель этого раздела заключается в том, чтобы показать, что такое ORM и как он работает на высоком уровне. Автор SQLAlchemy написал полное руководство к нему (http://bit.ly/obj-rel-tutorial). После прочтения этого раздела определитесь, какой из следующих уровней лучше подходит для ваших нужд:

- простой DB-API, показанный ранее в подразделе «SQLite»;
- движок SQLAlchemy;
- язык выражений SQLAlchemy;
- SQLAlchemy ORM.

Естественным выбором выглядит применение ORM, что позволит избежать всех сложностей SQL. Стоит ли им пользоваться? Некоторые люди считают, что ORM следует избегать (http://bit.ly/obj-rel-map), а другие полагают, что его критикуют незаслуженно (http://bit.ly/fowler-orm). Независимо от того, кто прав, ORM — это абстракция, а все абстракции в какой-то момент разрушаются — они допускают утечки памяти. Если ORM не делает того, что вам нужно, вы должны понять, как он работает, а затем разобраться, как исправить это с помощью SQL. Перефразируя интернет-мем, некоторые люди, столкнувшись с проблемой, думают: «Точно, использую ORM». Теперь у них две проблемы. Старайтесь использовать ORM реже и, как правило, для простых приложений. Но если приложение кажется простым, то вам, возможно, стоит использовать простой SQL (или язык выражений SQL).

Или же вы можете попробовать еще более простой способ — `dataset` (https://dataset.readthedocs.org/). Он создан на основе SQLAlchemy и предоставляет простой ORM для хранилищ SQL, JSON и CSV.

Хранилища данных NoSQL

Некоторые базы данных не являются реляционными и не поддерживают SQL. Они были созданы для работы с очень крупными наборами данных, позволяют более гибко определять данные и поддерживают пользовательские операции с данными. Такие базы данных называют NoSQL (раньше это означало «не SQL», теперь же расшифровка звучит как «не только SQL»).

Семейство dbm

Форматы dbm существовали задолго до того, как появился *NoSQL*. Они представляют собой хранилища, работающие по принципу «ключ — значение», их часто встраивают в приложения вроде браузеров, чтобы поддерживать различные настройки. База данных dbm очень похожа на обычный словарь.

- Вы присваиваете значение ключу, и оно автоматически сохраняется в базе данных на диске.
- Вы можете получить значение с помощью ключа.

Рассмотрим простой пример. Второй аргумент следующего метода open() может принимать значения 'r' для чтения, 'w' для записи и 'c' для того и другого, создавая файл, если его не существует:

```
>>> import dbm
>>> db = dbm.open('definitions', 'c')
```

Для того чтобы создать пары «ключ — значение», просто присвойте значение ключу, как если бы вы работали со словарем:

```
>>> db['mustard'] = 'yellow'
>>> db['ketchup'] = 'red'
>>> db['pesto'] = 'green'
```

Приостановимся и посмотрим, что мы уже имеем:

```
>>> len(db)
3
>>> db['pesto']
b'green'
```

Теперь закроем файл и откроем его снова, чтобы убедиться, что наши данные действительно были сохранены:

```
>>> db.close()
>>> db = dbm.open('definitions', 'r')
>>> db['mustard']
b'yellow'
```

Ключи и значения сохраняются как байты. Вы не можете итерировать по объектам базы данных `db`, но можете получить количество ключей с помощью функции `len()`. Обратите внимание на то, что функции `get()` и `setdefault()` работают точно так же, как и для словарей.

Memcached

memcached (http://memcached.org/) — это быстрый сервер *кэширования*, располагающийся в памяти и работающий по принципу «ключ — значение». Его часто размещают перед базой данных, также он может использоваться для хранения данных сессии веб-сервера. Вы можете загрузить версии для Linux, OS X (http://bit.ly/install-osx) и Windows (http://bit.ly/memcache-win). Если вы хотите попробовать запустить примеры, показанные в этом разделе, вам понадобятся сервер memcached и драйвер Python.

Существует множество драйверов Python, тот, что работает с Python 3, называется python3-memcached (https://github.com/eguven/python3-memcached), вы можете установить его с помощью этой команды:

```
$ pip install python-memcached
```

Для того чтобы использовать его, подключитесь к серверу memcached, после чего можете:

- устанавливать и получать значения ключей;
- увеличивать и уменьшать значения;
- удалять ключи.

Данные, хранимые в базе, неустойчивы, они могут исчезнуть. Это происходит из-за того, что memcached является сервером кэша. Он избегает ситуаций, когда у него заканчивается память, стирая старые данные.

Вы можете подключиться к нескольким серверам memcached одновременно. В следующем примере мы беседуем с одним и тем же компьютером:

```
>>> import memcache
>>> db = memcache.Client(['127.0.0.1:11211'])
>>> db.set('marco', 'polo')
True
>>> db.get('marco')
'polo'
>>> db.set('ducks', 0)
True
>>> db.get('ducks')
0
>>> db.incr('ducks', 2)
2
>>> db.get('ducks')
2
```

Redis

Redis — это *сервер структур данных*. Как и в случае с memcached, все данные сервера Redis должны поместиться в память (хотя у нас имеется возможность сохранить все данные на диск). В отличие от memcached Redis может делать следующее:

- сохранять данные на диск для надежности в случае перезагрузки;
- хранить старые данные;
- предоставлять более сложные структуры данных, нежели строки.

Типы данных, используемые Redis, похожи на типы данных, используемые в Python, и сервер Redis может быть применен в качестве промежуточного решения для того, чтобы одно или несколько приложений делились данными друг с другом. Я нахожу это настолько полезным, что посвящу этому небольшой фрагмент этой книги.

Исходный код драйвера Python `redis-py` и тесты находятся на GitHub (https://github.com/andymccurdy/redis-py), вы также можете найти документацию по нему (http://bit.ly/redis-py-docs). Можно установить этот драйвер с помощью следующей команды:

```
$ pip install redis
```

Сам по себе сервер Redis (http://redis.io/) хорошо задокументирован. Если вы установите и запустите его на своем локальном компьютере, который имеет сетевое имя `localhost`, можете попробовать запустить программы из следующих разделов.

Строки

Ключ, имеющий одно значение, является *строкой* Redis. Простые типы данных Python автоматически преобразовываются. Подключимся к серверу Redis, расположенному на некотором хосте (по умолчанию `localhost`) и порте (по умолчанию `6379`):

```
>>> import redis
>>> conn = redis.Redis()
```

Строки `redis.Redis('localhost')` или `redis.Redis('localhost', 6379)` дадут тот же результат.

Перечислим все ключи (которых пока нет):

```
>>> conn.keys('*')
[]
```

Создадим простую строку (с ключом `'secret'`), целое число (с ключом `'carats'`) и число с плавающей точкой (с ключом `'fever'`):

```
>>> conn.set('secret', 'ni!')
True
```

```
>>> conn.set('carats', 24)
True
>>> conn.set('fever', '101.5')
True
```

Получим значения согласно заданным ключам:

```
>>> conn.get('secret')
b'ni!'
>>> conn.get('carats')
b'24'
>>> conn.get('fever')
b'101.5'
```

Метод `setnx()` устанавливает значение, но только если ключа не существует:

```
>>> conn.setnx('secret', 'icky-icky-icky-ptang-zoop-boing!')
False
```

Метод не сработал, поскольку мы уже определили ключ `'secret'`:

```
>>> conn.get('secret')
b'ni!'
```

Метод `getset()` возвращает старое значение и одновременно устанавливает новое:

```
>>> conn.getset('secret', 'icky-icky-icky-ptang-zoop-boing!')
b'ni!'
```

Не будем сильно забегать вперед. Это сработало?

```
>>> conn.get('secret')
b'icky-icky-icky-ptang-zoop-boing!'
```

Теперь мы получим подстроку с помощью метода `getrange()` (как и в Python, смещение обозначается как `0` для начала списка и `-1` для конца):

```
>>> conn.getrange('secret', -6, -1)
b'boing!'
```

Заменим подстроку с помощью метода `setrange()` (используя смещение, которое начинается с нуля):

```
>>> conn.setrange('secret', 0, 'ICKY')
32
>>> conn.get('secret')
b'ICKY-icky-icky-ptang-zoop-boing!'
```

Далее установим значения сразу нескольких ключей с помощью метода `mset()`:

```
>>> conn.mset({'pie': 'cherry', 'cordial': 'sherry'})
True
```

Получим более одного значения с помощью метода mget():

```
>>> conn.mget(['fever', 'carats'])
[b'101.5', b'24']
```

Удалим ключ с помощью метода delete():

```
>>> conn.delete('fever')
True
```

Выполним инкремент с помощью команд incr() и incrbyfloat() и декремент с помощью команды decr():

```
>>> conn.incr('carats')
25
>>> conn.incr('carats', 10)
35
>>> conn.decr('carats')
34
>>> conn.decr('carats', 15)
19
>>> conn.set('fever', '101.5')
True
>>> conn.incrbyfloat('fever')
102.5
>>> conn.incrbyfloat('fever', 0.5)
103.0
```

Команды decrbyfloat() не существует. Используйте отрицательный инкремент, чтобы уменьшить значение ключа fever:

```
>>> conn.incrbyfloat('fever', -2.0)
101.0
```

Списки

Списки Redis могут содержать только строки. Список создается, когда вы добавляете первые данные. Добавим данные в начало списка с помощью метода lpush():

```
>>> conn.lpush('zoo', 'bear')
1
```

Добавим в начало списка более одного элемента:

```
>>> conn.lpush('zoo', 'alligator', 'duck')
3
```

Добавим один элемент до или после другого с помощью метода linsert():

```
>>> conn.linsert('zoo', 'before', 'bear', 'beaver')
4
>>> conn.linsert('zoo', 'after', 'bear', 'cassowary')
5
```

Добавим элемент, указав смещение для него, с помощью метода `lset()` (список уже должен существовать):

```
>>> conn.lset('zoo', 2, 'marmoset')
True
```

Добавим элемент в конец с помощью метода `rpush()`:

```
>>> conn.rpush('zoo', 'yak')
6
```

Получим элемент по заданному смещению с помощью метода `lindex()`:

```
>>> conn.lindex('zoo', 3)
b'bear'
```

Получим все элементы, находящиеся в диапазоне смещений, с помощью метода `lrange()` (можно использовать любой индекс от 0 до –1):

```
>>> conn.lrange('zoo', 0, 2)
[b'duck', b'alligator', b'marmoset']
```

Обрежем список с помощью метода `ltrim()`, сохранив только элементы в заданном диапазоне:

```
>>> conn.ltrim('zoo', 1, 4)
True
```

Получим диапазон значений (можно использовать любой индекс от 0 до –1) с помощью метода `lrange()`:

```
>>> conn.lrange('zoo', 0, -1)
[b'alligator', b'marmoset', b'bear', b'cassowary']
```

В главе 10 будет показано, как использовать списки Redis и механизм *публикации-подписки*, чтобы реализовать очереди задач.

Хеши

Хеши Redis похожи на словари в Python, но они могут содержать только строки. Поэтому вы можете создать только одномерный словарь. Рассмотрим примеры, в которых создается и изменяется хеш с именем `song`.

Установим в хеше `song` значения полей `do` и `re` одновременно с помощью метода `hmset()`:

```
>>> conn.hmset('song', {'do': 'a deer', 're': 'about a deer'})
True
```

Установим значение одного поля хеша с помощью метода `hset()`:

```
>>> conn.hset('song', 'mi', 'a note to follow re')
1
```

Получим значение одного поля с помощью метода hget():

```
>>> conn.hget('song', 'mi')
b'a note to follow re'
```

Получим значение нескольких полей с помощью метода hmget():

```
>>> conn.hmget('song', 're', 'do')
[b'about a deer', b'a deer']
```

Получим ключи всех полей хеша с помощью метода hkeys():

```
>>> conn.hkeys('song')
[b'do', b're', b'mi']
```

Получим значения всех полей хеша с помощью метода hvals():

```
>>> conn.hvals('song')
[b'a deer', b'about a deer', b'a note to follow re']
```

Получим количество полей хеша с помощью функции hlen():

```
>>> conn.hlen('song')
3
```

Получим ключи и значения всех полей хеша с помощью метода hgetall():

```
>>> conn.hgetall('song')
{b'do': b'a deer', b're': b'about a deer', b'mi': b'a note to follow re'}
```

Создадим поле, если его ключ не существует, с помощью метода hsetnx():

```
>>> conn.hsetnx('song', 'fa', 'a note that rhymes with la')
1
```

Множества

Множества Redis похожи на множества Python, как вы можете увидеть в следующих примерах.

Добавим одно или несколько значений множества:

```
>>> conn.sadd('zoo', 'duck', 'goat', 'turkey')
3
```

Получим количество значений множества:

```
>>> conn.scard('zoo')
3
```

Получим все значения множества:

```
>>> conn.smembers('zoo')
{b'duck', b'goat', b'turkey'}
```

Удалим значение из множества:

```
>>> conn.srem('zoo', 'turkey')
True
```

Создадим второе множество, чтобы продемонстрировать некоторые операции:

```
>>> conn.sadd('better_zoo', 'tiger', 'wolf', 'duck')
0
```

Пересечение множеств (получение общих членов) `zoo` и `better_zoo`:

```
>>> conn.sinter('zoo', 'better_zoo')
{b'duck'}
```

Выполним пересечение множеств `zoo` и `better_zoo` и сохраним результат в множестве `fowl_zoo`:

```
>>> conn.sinterstore('fowl_zoo', 'zoo', 'better_zoo')
1
```

Есть кто живой?

```
>>> conn.smembers('fowl_zoo')
{b'duck'}
```

Выполним объединение (всех членов) множеств `zoo` и `better_zoo`:

```
>>> conn.sunion('zoo', 'better_zoo')
{b'duck', b'goat', b'wolf', b'tiger'}
```

Сохраним результат этого пересечения в множестве `fabulous_zoo`:

```
>>> conn.sunionstore('fabulous_zoo', 'zoo', 'better_zoo')
4
>>> conn.smembers('fabulous_zoo')
{b'duck', b'goat', b'wolf', b'tiger'}
```

Какие элементы присутствуют в множестве `zoo` и отсутствуют в множестве `better_zoo`? Используйте метод `sdiff()`, чтобы получить разность множеств, и метод `sdiffstore()`, чтобы сохранить ее в множестве `zoo_sale`:

```
>>> conn.sdiff('zoo', 'better_zoo')
{b'goat'}
>>> conn.sdiffstore('zoo_sale', 'zoo', 'better_zoo')
1
>>> conn.smembers('zoo_sale')
{b'goat'}
```

Упорядоченные множества

Один из самых гибких типов данных Redis — это *упорядоченные множества*, или *zset*. Они представляют собой набор уникальных значений, но каждое значение связано

с дробным *счетчиком*. Вы можете получить доступ к каждому элементу с помощью его значения или счетчика. Упорядоченные множества применяются в качестве:

- списков лидеров;
- вторичных индексов;
- временных рядов, где отметки времени используются как счетчик.

Мы рассмотрим последний вариант применения, отслеживая логины пользователей с помощью временных меток. Мы будем использовать значение времени epoch (подробнее об этом — в главе 10), которое возвращает функция time():

```
>>> import time
>>> now = time.time()
>>> now
1361857057.576483
```

Добавим первого гостя (он немного нервничает):

```
>>> conn.zadd('logins', 'smeagol', now)
1
```

Пять минут спустя добавим второго гостя:

```
>>> conn.zadd('logins', 'sauron', now+(5*60))
1
```

Через два часа:

```
>>> conn.zadd('logins', 'bilbo', now+(2*60*60))
1
```

Еще один гость не торопился и пришел спустя сутки:

```
>>> conn.zadd('logins', 'treebeard', now+(24*60*60))
1
```

Каким по счету пришел bilbo?

```
>>> conn.zrank('logins', 'bilbo')
2
```

Когда это было?

```
>>> conn.zscore('logins', 'bilbo')
1361864257.576483
```

Посмотрим, каким по счету пришел каждый гость:

```
>>> conn.zrange('logins', 0, -1)
[b'smeagol', b'sauron', b'bilbo', b'treebeard']
```

И когда:

```
>>> conn.zrange('logins', 0, -1, withscores=True)
[(b'smeagol', 1361857057.576483), (b'sauron', 1361857357.576483),
(b'bilbo', 1361864257.576483), (b'treebeard', 1361943457.576483)]
```

Биты

Биты — это очень эффективный (с точки зрения занимаемого места) и быстрый способ обработать большое множество чисел. Предположим, у вас есть сайт, на котором регистрируются пользователи. Вы хотите отслеживать, как часто люди авторизуются, сколько пользователей посещает сайт в конкретный день, как часто один и тот же пользователь посещает сайт в следующие дни и т. д. Вы могли бы использовать множества Redis, но если вы присваиваете пользователям увеличивающиеся числовые ID, биты помогут вам быстрее и компактнее решить эту задачу.

Начнем с создания последовательности битов для каждого дня. Для этой проверки мы используем всего три дня и несколько ID:

```
>>> days = ['2013-02-25', '2013-02-26', '2013-02-27']
>>> big_spender = 1089
>>> tire_kicker = 40459
>>> late_joiner = 550212
```

Каждая дата является отдельным ключом. Установим бит для конкретного пользователя в эту дату. Например, в первую дату (`2013-02-25`) у нас есть посещения от `big_spender` (`ID 1089`) и `tire_kicker` (`ID 40459`):

```
>>> conn.setbit(days[0], big_spender, 1)
0
>>> conn.setbit(days[0], tire_kicker, 1)
0
```

На следующий день `big_spender` вернулся:

```
>>> conn.setbit(days[1], big_spender, 1)
0
```

На следующий день у нас снова появился наш друг `big_spender`, а также новый человек, которого мы назвали `late_joiner`:

```
>>> conn.setbit(days[2], big_spender, 1)
0
>>> conn.setbit(days[2], late_joiner, 1)
0
```

Получим счетчик ежедневных посещений за эти три дня:

```
>>> for day in days:
...     conn.bitcount(day)
...
2
1
2
```

Посещал ли сайт заданный пользователь в указанный день?

```
>>> conn.getbit(days[1], tire_kicker)
0
```

Значит, `tire_kicker` не посещал сайт во второй день.

Сколько пользователей посещает сайт каждый день?

```
>>> conn.bitop('and', 'everyday', *days)
68777
>>> conn.bitcount('everyday')
1
```

Угадайте с трех попыток, кто это:

```
>>> conn.getbit('everyday', big_spender)
1
```

Наконец, сколько уникальных пользователей посетили сайт за эти три дня?

```
>>> conn.bitop('or', 'alldays', *days)
68777
>>> conn.bitcount('alldays')
3
```

Кэши и истечение срока действия

У всех ключей Redis есть время жизни, или *дата истечения срока действия*. По умолчанию этот срок длится вечно. Мы можем использовать функцию `expire()`, чтобы указать Redis, как долго хранить заданный ключ. Как показано далее, значением является количество секунд:

```
>>> import time
>>> key = 'now you see it'
>>> conn.set(key, 'but not for long')
True
>>> conn.expire(key, 5)
True
>>> conn.ttl(key)
5
>>> conn.get(key)
b'but not for long'
>>> time.sleep(6)
>>> conn.get(key)
>>>
```

Команда `expireat()` указывает, что действие ключа истекает в заданное время эпохи Unix. Это может оказаться полезным для того, чтобы кэш оставался свежим и чтобы ограничить сессии авторизации.

Прочие серверы NoSQL

Серверы NoSQL, перечисленные здесь, могут работать с данными, объем которых превышает объем доступной памяти, и многие из них требуют использования нескольких компьютеров. В табл. 8.6 показаны наиболее популярные серверы и их библиотеки Python.

Таблица 8.6. Базы данных NoSQL

Сайт	Python API
Cassandra	pycassa
CouchDB	couchdb-python
HBase	happybase
Kyoto	kyotocabinet
MongoDB	mongodb
Riak	riak-python-client

Full-Text Databases

Наконец, существует особая категория баз данных для *полнотекстового* поиска. Они индексируют все, поэтому вы легко можете найти то стихотворение, в котором говорится о ветряных мельницах и гигантских головках сыра. Вы можете увидеть популярные примеры таких баз данных с открытым исходным кодом и их Python API в табл. 8.7.

Таблица 8.7. Полнотекстовые базы данных

Сайт	Python API
Lucene	pylucene
Solr	SolPython
ElasticSearch	pyes
Sphinx	sphinxapi
Xapian	xappy
Whoosh	Написан на Python, уже содержит API

Упражнения

1. Присвойте строку `'This is a test of the emergency text system'` переменной `test1` и запишите переменную `test1` в файл с именем `test.txt`.
2. Откройте файл `test.txt` и считайте его содержимое в строку `test2`. Совпадают ли строки `test1` и `test2`?
3. Сохраните следующие несколько строк в файл `books.csv`. Обратите внимание на то, что, если поля разделены запятыми, вам нужно заключить поле в кавычки, если оно содержит запятую:

   ```
   author,book
   J R R Tolkien,The Hobbit
   Lynne Truss,"Eats, Shoots & Leaves"
   ```

4. Используйте модуль csv и его метод DictReader, чтобы считать содержимое файла books.csv в переменную books. Выведите на экран значения переменной books. Обработал ли метод DictReader кавычки и запятые в заголовке второй книги?
5. Создайте CSV-файл books.csv и запишите его в следующие строки:

```
title,author,year
The Weirdstone of Brisingamen,Alan Garner,1960
Perdido Street Station,China Miéville,2000
Thud!,Terry Pratchett,2005
The Spellman Files,Lisa Lutz,2007
Small Gods,Terry Pratchett,1992
```

6. Используйте модуль sqlite3, чтобы создать базу данных SQLite books.db и таблицу books, содержащую следующие поля: title (text), author (text) и year (integer).
7. Считайте данные из файла books.csv и добавьте их в таблицу book.
8. Считайте и выведите на экран графу title таблицы book в алфавитном порядке.
9. Считайте и выведите на экран все графы таблицы book в порядке публикации.
10. Используйте модуль sqlalchemy, чтобы подключиться к базе данных sqlite3 books.db, которую вы создали в упражнении 6. Как и в упражнении 8, считайте и выведите на экран графу title таблицы book в алфавитном порядке.
11. Установите сервер Redis и библиотеку Python redis (с помощью команды pip install redis) на свой компьютер. Создайте хеш redis с именем test, содержащий поля count (1) и name ('Fester Bestertester'). Выведите все поля хеша test.
12. Увеличьте поле count хеша test и выведите его на экран.

9 Распутываем Всемирную паутину

На французско-швейцарской границе располагается CERN — Институт исследования физики частиц, он может показаться хорошим убежищем для злодея из франшизы о Джеймсе Бонде. К счастью, его задача заключается не в получении мирового господства, а в том, чтобы понять принципы работы Вселенной. Это всегда приводило к тому, что CERN генерировал удивительные объемы данных, заставляя физиков и компьютерщиков держать темп.

В 1989 году английский ученый Тим Бернерс-Ли (Tim Berners-Lee) впервые внес предложение помочь распространять информацию внутри CERN и исследовательского сообщества. Он назвал его *World Wide Web (Всемирная паутина)* и довольно быстро выделил три основные идеи, которые должны были лечь в основу ее дизайна:

- *HTTP (Hypertext Transfer Protocol, протокол передачи гипертекста)* — спецификация для веб-клиентов и серверов для обмена запросами и ответами;
- *HTML (Hypertext Markup Language, гипертекстовый язык разметки)* — формат для представления результатов;
- *URL (Uniform Resource Locator, единообразный локатор ресурса)* — способ уникально обозначить сервер и ресурс на этом сервере.

В самом простом варианте использования веб-клиент (я думаю, что Бернерс-Ли был первым, кто употребил слово «браузер») соединяется с веб-сервером с помощью протокола HTTP, запрашивает URL и получает HTML.

Он написал первый браузер и сервер на компьютере NeXT, изобретенном небольшой компанией, которую основал Стив Джобс (Steve Jobs) во время своего отдыха от Apple Computer. Известность Всемирной паутины значительно возросла в 1993-м, когда группа студентов Иллинойсского университета (University of Illinois) выпустила браузер Mosaic (для Windows, Macintosh и Unix) и сервер NCSA *httpd*. Когда я загрузил их и начал создавать сайты, я даже не догадывался, что Всемирная паутина и Интернет станут частью повседневной жизни. В то время Интернет все еще был некоммерческим официально, в мире существовало всего 500 известных веб-серверов (http://home.web.cern.ch/about/birth-web). К концу 1994 года их количество увеличилось до 10 000. Интернет был открыт для коммерческого

использования, и авторы браузера Mosaic основали компанию Netscape, чтобы писать коммерческие веб-приложения. Компания Netscape стала достоянием общественности как часть возникшего в то время интернет-безумия, и взрывной рост Всемирной паутины не остановился до сих пор.

Практически каждый язык программирования был использован для написания веб-клиентов и веб-серверов. Динамические языки Perl, PHP и Ruby стали особенно популярными. В этой главе я покажу вам, почему Python является особенно хорошим языком для работы в Интернете на любом из следующих уровней:

- клиенты для удаленного доступа;
- серверы, предоставляющие данные для сайтов и веб-API;
- веб-API и сервисы, позволяющие обмениваться данными другими способами, отличающимися от просматриваемых веб-страниц.

Выполняя упражнения в конце главы, мы создадим настоящий интерактивный сайт.

Веб-клиенты

Низкоуровневая система проводящих путей Интернета называется Transmission Control Protocol/Internet Protocol (протокол управления передачей/интернет-протокол), или просто TCP/IP (в подразделе «TCP/IP» раздела «Сети» главы 11 этот протокол рассматривается более подробно). Он перемещает байты между компьютерами, но не обращает внимания на то, что они значат. Это работа высокоуровневых *протоколов* — определений синтаксиса для некоторых целей. HTTP — это стандартный протокол для обмена данными в Сети.

Всемирная паутина — это клиент-серверная система. Клиент делает *запрос* серверу: он открывает соединение TCP/IP, отправляет URL и другую информацию с помощью HTTP и получает *ответ*.

Формат ответа также определяется протоколом HTTP. Он включает в себя статус запроса и (в том случае, если запрос выполнен успешно) данные и формат ответа.

Самый известный веб-клиент — это *браузер*. Он может создавать HTTP-запросы несколькими способами. Вы можете инициировать запрос вручную, написав URL в адресной строке или щелкнув на ссылке на веб-странице. Очень часто для отображения сайта используются возвращаемые данные: HTML-документы, файлы JavaScript, файлы CSS и изображения, — но данные могут быть любого типа, в том числе и не предназначенные для отображения.

Важный аспект HTTP — этот протокол *не имеет состояния*. Каждое создаваемое вами соединение HTTP не зависит от других. Это упрощает базовые операции, но усложняет другие. Рассмотрим несколько примеров таких усложнений.

- *Кэширование*. Удаленный контент, который не меняется, должен быть сохранен веб-клиентом и использован для того, чтобы не загружать его с сервера снова.

- *Сессии*. Интернет-магазин должен запоминать содержимое вашей корзины.
- *Аутентификация*. Сайты, которые требуют ваши имя пользователя и пароль, должны запоминать их, пока вы авторизованы.

Решения для этих усложнений включают в себя *cookie*, в которых сервер отправляет клиенту довольно специфическую информацию, позволяющую их распознать, когда клиент отправляет cookie назад.

Тестируем с telnet

HTTP — это протокол, основанный на тексте, поэтому вы можете вручную вводить его код во время тестирования. Древняя программа telnet позволяет вам подключиться к любому серверу и порту и вводить команды.

Запросим у любимого многими тестового сайта Google базовую информацию о его домашней странице. Введем следующее:

```
$ telnet www.google.com 80
```

Если на порте `80` по адресу `google.com` существует веб-сервер (я думаю, что это беспроигрышный вариант), telnet выведет на экран подтверждающую информацию, а затем отобразит пустую строку, которая является приглашением ввести что-то еще:

```
Trying 74.125.225.177...
Connected to www.google.com.
Escape character is '^]'.
```

Теперь введем настоящую команду HTTP для telnet, которую он отправит на веб-сервер Google. Самая распространенная команда HTTP (ее использует ваш браузер каждый раз, когда вы вводите URL в адресной строке) — это `GET`. Она позволяет получить содержимое заданного ресурса вроде HTML-файла и возвращает его клиенту. Для первой проверки мы используем команду HTTP `HEAD`, которая просто получает некую базовую информацию о ресурсе:

```
HEAD / HTTP/1.1
```

Конструкция `HEAD /` отправляет запрос HTTP `HEAD` *глагол* (команда), чтобы получить информацию о домашней странице (/). Добавьте дополнительный символ возврата каретки, чтобы отправить пустую строку, тогда удаленный сервер будет знать, что вы закончили и ждете ответа. Вы получите ответ вроде следующего (мы обрезали некоторые длинные строки с помощью многоточий, чтобы они не вываливались за пределы страницы):

```
HTTP/1.1 200 OK
Date: Sat, 26 Oct 2013 17:05:17 GMT
Expires: -1
Cache-Control: private, max-age=0
Content-Type: text/html; charset=ISO-8859-1
Set-Cookie: PREF=ID=962a70e9eb3db9d9:FF=0:TM=1382807117:LM=1382807117:S=y...
```

```
  expires=Mon, 26-Oct-2015 17:05:17 GMT;
  path=/;
  domain=.google.com
Set-Cookie: NID=67=hTvtVC7dZJmZzGktimbwVbNZxPQnaDijCz716B1L56GM9qvsqqeIGb...
  expires=Sun, 27-Apr-2014 17:05:17 GMT
Web Clients  |  219  path=/;
  domain=.google.com;
  HttpOnly
P3P: CP="This is not a P3P policy! See http://www.google.com/support/accounts...
Server: gws
X-XSS-Protection: 1; mode=block
X-Frame-Options: SAMEORIGIN
Alternate-Protocol: 80:quic
Transfer-Encoding: chunked
```

Так выглядят заголовки ответов HTTP и их значения. Некоторые из них, вроде `Date` или `Content-Type`, обязательны. Другие, наподобие `Set-Cookie`, используются для отслеживания вашей активности в течение нескольких посещений (мы поговорим об *управлении состоянием* немного позже). Когда вы делаете запрос HTTP `HEAD`, то получаете в ответ только заголовки. Если вы использовали команды HTTP `GET` или HTTP `POST`, также получите данные от домашней страницы (смесь HTML, CSS, JavaScript и всего прочего, что Google решит разместить на своей домашней странице).

Я не хочу, чтобы вы зависли в telnet. Чтобы его закрыть, введите следующее:

```
q
```

Стандартные веб-библиотеки Python

В Python 2 модули веб-клиентов и веб-серверов были слегка разбросаны. Одна из целей Python 3 заключается в том, чтобы разместить эти модули в двух *пакетах* (как вы помните из главы 5, пакет — это всего лишь папка для хранения файлов модулей).

- `http` управляет всеми деталями клиент-серверного взаимодействия HTTP:
  - `client` выполняет всю работу на стороне клиента;
  - `server` помогает вам написать веб-сервер;
  - `cookies` и `cookiejar` управляют cookies, которые сохраняют данные между посещениями;
- `urllib` работает на базе `http`:
  - `request` обрабатывает клиентские запросы;
  - `response` обрабатывает ответы сервера;
  - `parse` разбивает URL на части.

Воспользуемся стандартной библиотекой, чтобы получить что-нибудь с сайта. URL в следующем примере возвращает случайную текстовую цитату — это что-то вроде печенья с предсказанием:

```
>>> import urllib.request as ur
>>> url = 'http://www.iheartquotes.com/api/v1/random'
>>> conn = ur.urlopen(url)
>>> print(conn)
<http.client.HTTPResponse object at 0x1006fad50>
```

Из официальной документации (http://bit.ly/httpresponse-docs) мы можем узнать, что `conn` является объектом класса `HTTPResponse`, содержащим несколько методов, и его метод `read()` предоставит нам информацию о веб-странице:

```
>>> data = conn.read()
>>> print(data)
b'You will be surprised by a loud noise.\r\n\n[codehappy]
http://iheartquotes.com/fortune/show/20447\n'
```

Этот небольшой фрагмент кода открыл соединение TCP/IP с удаленным сервером цитат, создал запрос HTTP и получил HTTP-ответ. Ответ содержит не только данные о странице (цитату). Одна из наиболее важных частей ответа — это *код статуса* HTTP:

```
>>> print(conn.status)
200
```

Значение `200` означает, что все прошло гладко. Существуют десятки кодов статуса HTTP, объединенных в пять диапазонов в соответствии с их первой цифрой (сотни):

- 1xx *(информация)*. Сервер получил запрос, но имеет некоторую дополнительную информацию для клиента;
- 2xx *(успех)*. Сработало, каждый код успеха, кроме 200, сообщает дополнительные детали;
- 3xx *(перенаправление)*. Ресурс был перемещен, поэтому ответ возвращает клиенту новый URL;
- 4xx *(ошибка клиента)*. Некоторые проблемы на стороне клиента вроде знаменитой ошибки 404 (ресурс не найден). Код 418 *(I'm a teapot)* был первоапрельской шуткой;
- 5xx *(ошибка сервера)*. Код 500 — это общая ошибка. Вы можете встретить ошибку 502 (ошибочный шлюз), если произошел разрыв связи между веб-сервером и машинным интерфейсом.

Веб-серверы могут отправлять данные назад в том формате, который им нравится. Обычно это HTML (а также немного CSS и JavaScript), но в нашем примере с печеньем с предсказанием это простой текст. Формат данных указывается

значением *заголовка* ответа HTTP Content-Type, который мы также видели в примере с google.com:

```
>>> print(conn.getheader('Content-Type'))
text/plain
```

Строка text/plain является *MIME-типом* и означает, что данные пришли в простом текстовом формате. MIME-тип для HTML, который отправил пример с google.com, — это text/html. В этой главе я покажу вам еще несколько MIME-типов.

Из любопытства взглянем, какие еще заголовки HTTP были нам отправлены:

```
>>> for key, value in conn.getheaders():
...     print(key, value)
...
Server nginx
Date Sat, 24 Aug 2013 22:48:39 GMT
Content-Type text/plain
Transfer-Encoding chunked
Connection close
Etag "8477e32e6d053fcfdd6750f0c9c306d6"
X-Ua-Compatible IE=Edge,chrome=1
X-Runtime 0.076496
Cache-Control max-age=0, private, must-revalidate
```

Помните тот пример работы с telnet, который я показывал ранее? Теперь наша библиотека Python может разбирать заголовки этих HTTP-запросов и размещать их в словарь. Date и Server кажутся довольно очевидными, некоторые другие — нет. Полезно знать, что HTTP имеет набор стандартных заголовков вроде Content-Type и множество опциональных.

За пределами стандартной библиотеки: requests

В начале главы 1 вы увидели программу, которая получает доступ к YouTube API с помощью стандартных библиотек urllib.request и json. После него был другой пример, который использовал стороннюю библиотеку requests. Он был короче и проще для понимания.

Я считаю, что для большинства задач, связанных с разработкой веб-клиентов, проще использовать библиотеку requests. Вы можете просмотреть ее документацию по адресу http://docs.python-requests.org/ (она довольно хорошо написана), чтобы получить более подробную информацию. Я покажу вам основные принципы работы с этой библиотекой в данном разделе и буду использовать ее на протяжении всей книги для решения задач, связанных с веб-клиентами.

Для начала вам нужно установить библиотеку requests в свое окружение Python. Из окна терминала (пользователи Windows должны ввести cmd, чтобы получить

к нему доступ) введите следующую команду, чтобы установщик пакетов Python pip загрузил последнюю версию пакета и установил ее:

```
$ pip install requests
```

Если у вас возникли трудности, прочтите приложение Г, чтобы узнать подробности о том, как установить и использовать `pip`.

Переделаем предыдущий вызов сервиса с цитатами с помощью библиотеки requests:

```
>>> import requests
>>> url = 'http://www.iheartquotes.com/api/v1/random'
>>> resp = requests.get(url)
>>> resp
<Response [200]>
>>> print(resp.text)
I know that there are people who do not love their fellow man, and I hate
people like that!
    -Tom Lehrer, Satirist and Professor
 [codehappy] http://iheartquotes.com/fortune/show/21465
```

Этот пример не сильно отличается от предыдущего, где использовалась библиотека urllib.request.urlopen, но он кажется чуть менее объемным.

Веб-серверы

Веб-разработчики обнаружили, что Python хорошо подходит для написания веб-серверов и программ, работающих на серверной стороне. Это привело к появлению такого множества *фреймворков*, написанных на этом языке, что теперь уже становится трудно исследовать их все и сделать выбор, не говоря уже о том, чтобы решить, о каких из них поговорить в книге.

Веб-фреймворк предоставляет функции, с помощью которых вы можете построить сайты, поэтому он может решать большее количество задач, чем простой веб-сервер (HTTP). Вы встретитесь с функциями маршрутизации (URL к функции сервера), шаблонами (HTM с динамическими включениями), отладкой и др.

Я не буду говорить в этой книге обо всех фреймворках — рассмотрю лишь те, которые относительно просты в использовании и подходят для создания настоящих сайтов. Я также покажу вам, как запускать динамические части сайта с помощью Python и других составляющих на традиционном веб-сервере.

Простейший веб-сервер Python

Вы можете запустить простейший веб-сервер, просто введя одну строку кода Python:

```
$ python -m http.server
```

С помощью этой строки вы реализуете примитивный Python HTTP server. Если никаких проблем не возникло, вы увидите исходное сообщение о статусе:

```
Serving HTTP on 0.0.0.0 port 8000 ...
```

Запись `0.0.0.0` означает *любой адрес TCP*, поэтому веб-клиенты могут получать к нему доступ независимо от того, какой адрес имеет сервер. В главе 11 вы можете прочитать о некоторых низкоуровневых деталях TCP и других системах соединения в сеть.

Теперь вы можете запрашивать файлы, чьи пути относительны к вашему текущему каталогу, и они будут вам возвращены. Если вы введете в своем браузере строку `http://localhost:8000`, то должны увидеть список каталогов, и сервер выведет на экран строки обращения к журналам наподобие следующих:

```
127.0.0.1 -- [20/Feb/2013 22:02:37] "GET / HTTP/1.1" 200 -
```

`localhost` и `127.0.0.1` являются для TCP синонимами *вашего локального компьютера*, поэтому они сработают независимо от того, подключены ли вы к Интернету. Вы можете интерпретировать эти строки следующим образом.

- `127.0.0.1` — это IP-адрес клиента.
- Первый символ `-` — это имя удаленного пользователя, если он присутствует.
- Второй символ `-` — это имя авторизующегося пользователя, если требуется.
- `[20/Feb/2013 22:02:37]` — это дата и время доступа.
- `"GET / HTTP/1.1"` — это команда, отправленная веб-серверу:
  - метод HTTP (`GET`);
  - запрошенный ресурс (`/`, верхний уровень);
  - версия HTTP (`HTTP/1.1`).
- Последнее число (`200`) — это код статуса HTTP, возвращенный веб-сервером.

Щелкните на любом файле. Если ваш браузер может распознать его формат (HTML, PNG, GIF, JPEG и т. д.), он должен отобразить его, и сервер занесет этот запрос в журнал. Например, если в вашем текущем каталоге имеется файл `oreilly.png`, запрос `http://localhost:8000/oreilly.png` должен вернуть изображение встревоженной зверушки, показанное на рис. 7.1, а в журнале должна появиться похожая запись:

```
127.0.0.1 -- [20/Feb/2013 22:03:48] "GET /oreilly.png HTTP/1.1" 200 -
```

Если у вас в этой папке находятся и другие файлы, их названия должны появиться в списке. Можете щелкнуть на одном из файлов, чтобы загрузить его. Если ваш браузер сконфигурирован так, чтобы отображать формат этого файла, вы увидите результат на экране, в противном случае браузер спросит, хотите ли вы загрузить и сохранить файл.

По умолчанию используется порт 8000, но вы можете указать любой другой:

```
$ python -m http.server 9999
```

Вы должны увидеть следующее:

```
Serving HTTP on 0.0.0.0 port 9999 ...
```

Этот сервер, написанный только на Python, лучше всего подходит для быстрых тестов. Мы можете выключить его, остановив его процесс с помощью комбинации клавиш Ctrl+C.

Вы не должны использовать этот простой сервер для загруженного производственного сайта. Традиционные веб-серверы вроде Apache и Nginx гораздо быстрее работают со статическими файлами. Кроме того, этот простой сервер не может работать с динамическим содержимым, на что оказываются способны более продвинутые серверы, принимая дополнительные параметры.

Web Server Gateway Interface

Довольно быстро необходимость в простых файлах исчезает, и нам уже нужен сервер, который может запускать программы динамически. В первые годы существования Всемирной паутины *общий интерфейс шлюза* (Common Gateway Interface, CGI) был разработан для того, чтобы веб-серверы могли запускать внешние программы и возвращать результаты. CGI также обрабатывал получение входных аргументов от клиента, передавая их через сервер сторонним программам. Однако программы запускались заново при каждом обращении клиента. Масштабировать такие системы было трудно, поскольку даже у небольших программ время загрузки довольно велико.

Для того чтобы избежать задержки запуска, люди начали встраивать интерпретатор языка в веб-сервер. Apache запускал код на PHP внутри своего модуля `mod_php`, Perl — внутри модуля `mod_perl` и Python — внутри модуля `mod_python`. Далее код этих динамических языков мог быть выполнен внутри долгоиграющего процесса Apache, а не во внешних программах.

Альтернативный метод заключается в том, чтобы запускать динамический язык внутри отдельной долгоиграющей программы и заставить ее обмениваться данными с веб-сервером. Примерами таких программ являются FastCGI и SCGI.

Веб-разработка с использованием Python совершила рывок с появлением *Web Server Gateway Interface* (WSGI) — универсального API между веб-приложениями и веб-серверами. Все веб-фреймворки и веб-серверы Python, показанные далее, используют WSGI. Обычно вам не нужно знать, как работает WSGI (для этого многого и не потребуется), но осведомленность об основных принципах его функционирования может действительно помочь разработке.

Фреймворки

Веб-серверы обрабатывают детали работы HTTP и WSGI, но вам нужно использовать веб-фреймворки для того, чтобы написать код Python, который будет поддерживать сайт. Поэтому сейчас мы немного поговорим о фреймворках, а затем

вернемся к альтернативным способам обслуживания сайтов, которые их используют.

Для того чтобы написать сайт с помощью Python, существует множество веб-фреймворков (некоторые даже могут сказать, что их слишком много). Веб-фреймворк обрабатывает как минимум запросы клиента и ответы сервера. Он может предоставлять следующие возможности:

- *маршруты* — интерпретирует URL и находит соответствующие файлы на сервере или серверный код Python;
- *шаблоны* — объединяет серверные данные в страницы HTML;
- *аутентификация и авторизация* — обрабатывает имена пользователей, пароли, разрешения;
- *сессии* — обслуживает временное хранилище данных во время посещения сайта пользователем.

В следующих разделах мы напишем пример, использующий два фреймворка (Bottle и Flask). Далее поговорим об альтернативах, в частности о сайтах, работающих с базами данных. Вы можете найти подходящий фреймворк Python для любого сайта, который только можете себе представить.

Bottle

Bottle состоит из одного файла Python, поэтому его довольно легко опробовать и развернуть. Bottle не является частью стандартной библиотеки Python, поэтому установите его с помощью следующей команды:

```
$ pip install bottle
```

Рассмотрим код, который запустит тестовый веб-сервер и вернет текстовую строку, когда ваш браузер обратится по URL http://localhost:9999/. Сохраните этот файл как `bottle1.py`:

```
from bottle import route, run
@route('/')
def home():
  return "It isn't fancy, but it's my home page"
run(host='localhost', port=9999)
```

Bottle использует декоратор `route`, чтобы связать URL со следующей функцией; в этом примере / (домашняя страница) обрабатывается функцией `home()`. Запустите этот сценарий сервера с помощью следующей команды:

```
$ python bottle1.py
```

Когда вы обратитесь по адресу http://localhost:9999, вы должны увидеть следующее:

```
It isn't fancy, but it's my home page
```

Функция `run()` запускает встроенный тестовый веб-сервер Bottle. Вам не нужно использовать его в программах, написанных с помощью Bottle, но это может оказаться полезным на первых этапах разработки и тестирования.

Теперь вместо создания текста домашней страницы в коде создадим отдельный HTML-файл, который называется `index.html` и содержит такую строку:

```
My <b>new</b> and <i>improved</i> home page!!!
```

Укажите Bottle возвращать содержимое этого файла, когда запрашивается домашняя страница. Сохраните этот сценарий как `bottle2.py`:

```
from bottle import route, run, static_file
@route('/')
def main():
    return static_file('index.html', root='.')
run(host='localhost', port=9999)
```

В вызове `static_file()` мы хотим получить файл `index.html` из каталога, указанного в `root` (в нашем случае в `'.'`, текущем каталоге). Если код предыдущего примера все еще выполняется, то остановите его. Теперь запустите новый сервер:

```
$ python bottle2.py
```

Каждый раз, когда вы обращаетесь к странице http:/localhost:9999/, вы должны видеть следующее:

```
My new and improved home page!!!
```

Добавим последний пример, который демонстрирует, как передавать аргументы в URL и использовать их. Конечно же, этот файл будет называться `bottle3.py`:

```
from bottle import route, run, static_file
@route('/')
def home():
    return static_file('index.html', root='.')
@route('/echo/<thing>')
def echo(thing):
    return "Say hello to my little friend: %s!" % thing
run(host='localhost', port=9999)
```

У нас появилась новая функция `echo()`, в которую мы хотим передавать строковый аргумент через URL. За это отвечает строка `@route('/echo/<thing>')` в предыдущем примере. Конструкция `<thing>` в маршруте означает, что все, что находится в URL после `/echo/`, присваивается строковому аргументу `thing`, который передается функции `echo`. Чтобы увидеть, что случится, остановите старый сервер, если он все еще работает, и запустите его с новым кодом:

```
$ python bottle3.py
```

Далее перейдите в браузере по ссылке http://localhost:9999/echo/Mothra. Вы должны увидеть следующее:

```
Say hello to my little friend: Mothra!
```

Оставьте `bottle3.py` работать еще на пару минут, чтобы мы могли попробовать что-нибудь еще. Вы проверяли, что эти примеры работают, вводя URL в браузер и глядя на отображаемые страницы. Вы также можете использовать клиентские библиотеки вроде requests, чтобы они выполняли работу за вас. Сохраните этот код как `bottle_test.py`:

```
import requests
resp = requests.get('http://localhost:9999/echo/Mothra')
if resp.status_code == 200 and \
  resp.text == 'Say hello to my little friend: Mothra!':
    print('It worked! That almost never happens!')
else:
    print('Argh, got this:', resp.text)
```

Отлично! Теперь запустите этот код:

```
$ python bottle_test.py
```

В терминале вы должны увидеть следующее:

```
It worked! That almost never happens!
```

Перед вами небольшой пример *юнит-теста*. В главе 12 вы можете получить более подробную информацию о том, почему тесты — это хорошо и как написать их с помощью Python.

У фреймворка Bottle больше возможностей, чем я вам показал. В частности, когда вызываете функцию `run()`, можете попробовать добавить следующие аргументы:

- `debug=True` — создает страницу отладки, если вы получаете ошибку HTTP;
- `reloader=True` — перезагружает страницу в браузере, если вы измените хотя бы небольшой кусочек кода.

Все это хорошо задокументировано на сайте разработчика http://bottlepy.org/docs/dev/.

Flask

Bottle — это хороший фреймворк для того, чтобы начать работу. Но если вам нужно больше возможностей, попробуйте Flask. Он был создан в 2010 году как первоапрельская шутка, но реакция энтузиастов вдохновила его автора, Армина Ронахера (Armin Ronacher), на то, чтобы сделать его настоящим фреймворком. Он назвал результат Flask («склянка»), обыгрывая название Bottle — «бутылка».

Flask в использовании почти так же прост, как и Bottle, но он поддерживает множество расширений, которые могут оказаться полезными в профессиональной веб-разработке, например аутентификацию с помощью Facebook и интеграцию с базами данных. Этот фреймворк мне нравится больше других веб-фреймворков Python, поскольку в нем сбалансированы простота использования и богатый набор функций.

Пакет Flask включает в себя библиотеку package WSGI werkzeug и библиотеку шаблонов jinja2. Вы можете установить его с помощью терминала:

```
$ pip install flask
```

Переделаем наш последний пример с использованием фреймворка Flask. Однако для начала нам нужно внести несколько изменений.

Во Flask папка по умолчанию для статических файлов называется `static`, и URL для таких файлов тоже начинается со `/static`. Мы изменяем папку на `'.'` (текущая папка) и префикс URL на `''` (пустой), чтобы позволить URL `/` отображать файл `index.html`.

В функции `run()` установка параметра `debug=True` активизирует также автоматическую перезагрузку, тогда как фреймворк Bottle для отладки и перезагрузки использует отдельные аргументы.

Сохраните этот код в файл `flask1.py`:

```
from flask import Flask
app = Flask(__name__, static_folder='.', static_url_path='')
@app.route('/')
def home():
    return app.send_static_file('index.html')
@app.route('/echo/<thing>')
def echo(thing):
    return "Say hello to my little friend: %s" % thing
app.run(port=9999, debug=True)
```

Далее запустите сервер из терминала или окна:

```
$ python flask1.py
```

Протестируйте домашнюю страницу, введя в браузер следующий URL:

```
http://localhost:9999/
```

Вы должны увидеть следующее (как и в случае с Bottle):

```
My new and improved home page!!!
```

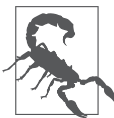
Попробуйте обратиться к конечной точке `/echo`:

```
http://localhost:9999/echo/Godzilla
```

Вы должны увидеть следующее:

```
Say hello to my little friend: Godzilla
```

Есть еще одно преимущество установки параметра `debug` равным `True` при вызове метода `run`. Если в серверном коде генерируется исключение, Flask возвращает особую отформатированную страницу, содержащую полезные сведения о том, что и где пошло не так. Даже больше: вы можете вводить команды, чтобы увидеть значения переменных в программе сервера.



Не устанавливайте параметр debug = True на производственных веб-серверах. Это предоставит потенциальным злоумышленникам слишком много информации о вашем сервере.

До сих пор примеры с использованием Flask повторяли то, что мы делали с помощью фреймворка Bottle. Что такого может делать Flask, чего не может делать Bottle? Flask содержит `jinja2` — более широкую систему шаблонов. Рассмотрим небольшой пример одновременного использования `jinja2` и `flask`.

Создайте папку `templates` и файл `flask2.html` внутри нее:

```
<html>
<head>
<title>Flask2 Example</title>
</head>
<body>
Say hello to my little friend: {{ thing }}
</body>
</html>
```

Далее мы напишем серверный код, который получает этот шаблон, заполняет значение аргумента `thing`, который мы передаем, и отрисовывает его как HTML (я опущу функцию `home()` для экономии места). Сохраните этот файл под именем `flask2.py`:

```
from flask import Flask, render_template
app = Flask(__name__)
@app.route('/echo/<thing>')
def echo(thing):
    return render_template('flask2.html', thing=thing)
app.run(port=9999, debug=True)
```

Аргумент `thing = thing` означает, что для передачи переменной с именем `thing` в шаблон эта переменная содержит значение строки `thing`.

Убедитесь, что файл `flask1.py` перестал работать, и запустите файл `flask2.py`:

```
$ python flask2.py
```

Теперь введите этот URL:

```
http://localhost:9999/echo/Gamera
```

Вы должны увидеть следующее:

```
Say hello to my little friend: Gamera
```

Модифицируем наш пример и сохраним его в папке `templates` под именем `flask3.html`:

```
<html>
<head>
<title>Flask3 Example</title>
</head>
<body>
Say hello to my little friend: {{ thing }}.
Alas, it just destroyed {{ place }}!
</body>
</html>
```

Второй аргумент в URL, `echo`, вы можете передать множеством способов.

Передача аргумента как части пути URL

С помощью этого метода вы просто расширяете URL (сохраните этот файл как `flask3a.py`):

```
from flask import Flask, render_template
app = Flask(__name__)
@app.route('/echo/<thing>/<place>')
def echo(thing, place):
    return render_template('flask3.html', thing=thing, place=place)
app.run(port=9999, debug=True)
```

Как обычно, остановите предыдущий сценарий тестового сервера, если он еще работает, и затем запустите новый:

```
$ python flask3a.py
```

URL должен выглядеть так:

```
http://localhost:9999/echo/Rodan/McKeesport
```

Вы должны увидеть следующее:

```
Say hello to my little friend: Rodan. Alas, it just destroyed McKeesport!
```

Или же вы можете передать аргументы как параметры команды `GET` (сохраните файл как `flask3b.py`):

```
from flask import Flask, render_template, request
app = Flask(__name__)
@app.route('/echo/')
def echo():
    thing = request.args.get('thing')
```

```
    place = request.args.get('place')
    return render_template('flask3.html', thing=thing, place=place)
app.run(port=9999, debug=True)
```

Запустите новый сценарий сервера:

```
$ python flask3b.py
```

В этот раз используйте следующий URL:

```
http://localhost:9999/echo?thing=Gorgo&place=Wilmerding
```

Вы должны увидеть следующее:

```
Say hello to my little friend: Gorgo. Alas, it just destroyed Wilmerding!
```

Когда команда `GET` используется в URL, любые аргументы должны передаваться в формате `&key1=val1&key2=val2&...`

Вы также можете использовать оператор словаря **, чтобы передать несколько аргументов в шаблон с помощью одного словаря (назовите файл `flask3c.py`):

```
from flask import Flask, render_template, request
app = Flask(__name__)
@app.route('/echo/')
def echo():
    kwargs = {}
    kwargs['thing'] = request.args.get('thing')
    kwargs['place'] = request.args.get('place')
    return render_template('flask3.html', **kwargs)
app.run(port=9999, debug=True)
```

`**kwargs` действует как конструкция `thing=thing, place=place`. Используя этот словарь, можно сэкономить немного времени, если входных аргументов много.

Язык шаблонов `jinja2` способен на гораздо большее. Если вы работали на PHP, то увидите много похожих возможностей.

Веб-серверы, не использующие Python

До этого момента мы использовали простые веб-серверы: `http.server` из стандартной библиотеки или сервера отладки Bottle и Flask. На производстве вам нужно запускать код на более быстрых серверах. Как правило, вы выбираете один из следующих вариантов:

- Apache с модулем `mod_wsgi`;
- Nginx с сервером приложений `uWSGI`.

Оба они работают хорошо: Apache, скорее всего, более популярен, а у Nginx имеется репутация стабильного и тратящего меньше памяти сервера.

Apache

Лучшим WSGI-модулем Apache (http://httpd.apache.org/) является mod_wsgi (https://code.google.com/p/modwsgi/). Он может запускать код, написанный на Python, внутри процесса Apache или в отдельном процессе, который обменивается данными с Apache.

Если вы используете Linux или OS X, в вашей системе Apache уже установлен. Для Windows вам придется устанавливать Apache самостоятельно (http://bit.ly/apache-http).

Наконец, установите предпочитаемый веб-фреймворк Python, основанный на WSGI. Попробуем использовать в наших примерах фреймворк Bottle. Практически вся работа включает в себя конфигурирование Apache, что может оказаться довольно затруднительным.

Создайте тестовый файл и сохраните его как /var/www/test/home.wsgi:

```
import bottle
application = bottle.default_app()
@bottle.route('/')
def home():
    return "apache and wsgi, sitting in a tree"
```

В этот раз не вызывайте функцию run(), поскольку это запустит встроенный веб-сервер Python. Нам нужно присвоить некоторое значение переменной application, поскольку именно его будет проверять mod_wsgi при объединении веб-сервера и кода Python.

Если Apache и его модуль mod_wsgi работают корректно, нужно лишь соединить их с нашим сценарием Python. Нам нужно добавить в файл одну строку, которая определяет сайт по умолчанию для этого сервера Apache, но поиск этого файла сам по себе является задачей. Он может называться /etc/apache2/httpd.conf, или /etc/apache2/sites-available/default, или даже быть латинским названием чьей-то ручной саламандры.

Предположим, что вы понимаете работу Apache и нашли нужный файл. Добавьте эту строку в раздел <VirtualHost>, который управляет стандартным сайтом:

```
    WSGIScriptAlias / /var/www/test/home.wsgi
```

Этот раздел должен выглядеть так:

```
<VirtualHost *:80>
    DocumentRoot /var/www
    WSGIScriptAlias / /var/www/test/home.wsgi
    <Directory /var/www/test>
    Order allow,deny
    Allow from all
    </Directory>
</VirtualHost>
```

Запустите Apache или перезапустите его, если работал, чтобы указать ему, что следует использовать новую конфигурацию. Если вы перейдете в браузере по адресу `http://localhost/`, то должны увидеть эту строку:

```
apache and wsgi, sitting in a tree
```

Это запустит `mod_wsgi` во *встроенном режиме* как часть самого Apache.

Вы также можете запустить его в *режиме демона* — как один или несколько процессов, отдельных от Apache. Для того чтобы это сделать, добавьте две новые строки директив в ваш файл конфигурации Apache:

```
$ WSGIDaemonProcess domain-name user=user-name group=group-name threads=25
WSGIProcessGroup domain-name
```

В предыдущем примере переменные `user-name` и `group-name` представляют собой имена пользователя и группы в операционной системе, а переменная `domain-name` — имя вашего интернет-домена. Минимальная конфигурация Apache может выглядеть так:

```
<VirtualHost *:80>
    DocumentRoot /var/www
    WSGIScriptAlias / /var/www/test/home.wsgi
    WSGIDaemonProcess mydomain.com user=myuser group=mygroup threads=25
    WSGIProcessGroup mydomain.com
    <Directory /var/www/test>
    Order allow,deny
    Allow from all
    </Directory>
</VirtualHost>
```

Веб-сервер Nginx

Веб-сервер Nginx не имеет встроенного модуля Python. Вместо этого он обменивается данными с помощью отдельного сервера WSGI вроде uWSGI. Вместе они представляют собой очень быструю и удобную в конфигурации платформу для веб-разработки с помощью Python.

Вы можете установить Nginx с его официального сайта http://wiki.nginx.org/Install. Вам также нужно установить uWSGI (http://bit.ly/uWSGI). uWSGI — это крупная система, имеющая множество различных настроек. Небольшая страница документации предоставляет вам инструкции, позволяющие объединить Flask, Nginx и uWSGI.

Другие фреймворки

Сайты и базы данных похожи на арахисовое масло и желе — часто можно увидеть, как они работают вместе. Небольшие фреймворки вроде Bottle и Flask не включают в себя функции поддержки баз данных, хотя некоторые надстройки их имеют.

Если вам нужно поставить на поток производство сайтов, работающих с базой данных, а сама база меняется не очень часто, можете попробовать воспользоваться одним из более крупных фреймворков. Рассмотрим самые известные из них.

- *django* (https://www.djangoproject.com/). Этот фреймворк самый популярный, особенно для крупных сайтов. Его стоит изучить по многим причинам, среди которых регулярно появляющиеся требования опыта работы с django в объявлениях о вакансиях. Он содержит код ORM (об ORM мы говорили в пункте «The Object-Relational Mapper» подраздела «SQLAlchemy» раздела «Реляционные базы данных» главы 8), позволяющий создавать автоматические веб-страницы для типичных функций баз данных CRUD (создание, замена, обновление, удаление), которые я рассматривал в подразделе «SQL» раздела «Реляционные базы данных» главы 8. Вам не обязательно использовать ORM именно для django, если больше нравится применять что-то другое, например SQLAlchemy или прямые запросы SQL.
- *web2py* (http://www.web2py.com/). Он работает примерно с тем же, с чем и django, только немного в другом стиле.
- *pyramid* (http://www.pylonsproject.org/). Этот фреймворк появился из более раннего проекта pylons, с точки зрения функционала он похож на django.
- *turbogears* (http://turbogears.org/). Этот фреймворк поддерживает ORM, множество баз данных и несколько языков шаблонов.
- *wheezy.web* (http://pythonhosted.org/wheezy.web/). Этот более молодой фреймворк оптимизирован для повышения производительности. Недавние исследования показали, что он работает быстрее других.

Вы можете сравнить фреймворки с помощью онлайн-таблицы по адресу http://bit.ly/web-frames.

Если вы хотите создать сайт, работающий с реляционной базой данных, вам не обязательно пользоваться одним из этих крупных фреймворков. Можете воспользоваться Bottle, Flask или каким-нибудь другим простым фреймворком и модулями работы с базами данных вроде SQLAlchemy, чтобы сгладить разногласия. Далее вам нужно будет написать обычный код SQL вместо специфического кода ORM, так как большинство разработчиков знают SQL, а не некий определенный синтаксис ORM.

Кроме того, ничто не заставляет вас выбирать именно реляционную базу данных. Если схема ваших данных может значительно изменяться — графы явно различаются в разных рядах, — вам стоит выбрать базу данных, *не имеющую схемы*, вроде баз данных NoSQL, которые мы рассматривали в разделе «Хранилища данных NoSQL» главы 8. Однажды я работал над сайтом, который изначально хранил данные в базе данных NoSQL, затем переключался на одну реляционную базу данных, затем на другую реляционную базу данных, затем на другую базу данных NoSQL и, наконец, возвращался к одной из реляционных.

Другие веб-серверы Python

Далее перечислены некоторые независимые WSGI-серверы, написанные на Python, которые работают как Apache или Nginx и используют несколько процессов и/или потоков (смотрите раздел «Конкуренция» главы 11) для обработки одновременных запросов:

- uwsgi (http://projects.unbit.it/uwsgi/);
- cherrypy (http://www.cherrypy.org/);
- pylons (http://www.pylonsproject.org/).

Далее перед вами серверы, *основанные на событиях*, которые пользуются одним процессом, но избегают блокирования любым одиночным запросом:

- tornado (http://www.tornadoweb.org/);
- gevent (http://gevent.org/);
- gunicorn (http://gunicorn.org/).

О событиях я поговорю подробнее в разделе о *конкуренции* в главе 11.

Веб-сервисы и автоматизация

Только что мы рассмотрели традиционные веб-клиенты и серверные приложения, потребляющие и генерирующие HTML-страницы. Всемирная паутина оказалась мощным способом объединять приложения и данные во многих форматах, не только HTML.

Модуль webbrowser

Начнем с небольшого сюрприза. Запустите сессию Python в окне терминала и введите следующую строку:

```
>>> import antigravity
```

Эта строка скрыто вызывает модуль стандартной библиотеки webbrowser и перенаправляет ваш браузер по просветительской ссылке. (Если вы по какой-то причине не видите ее, посетите сайт xkcd.)

Вы можете использовать этот модуль непосредственно. Эта программа загружает страницу главного сайта о Python в ваш браузер:

```
>>> import webbrowser
>>> url = 'http://www.python.org/'
>>> webbrowser.open(url)
True
```

Этот код откроет ее в новом окне:

```
>>> webbrowser.open_new(url)
True
```

А этот — на новой вкладке, если ваш браузер поддерживает вкладки:

```
>>> webbrowser.open_new_tab('http://www.python.org/')
True
```

Модуль webbrowser заставляет браузер делать всю работу.

API для Сети и Representational State Transfer

Зачастую данные доступны только внутри веб-страниц. Если вы хотите получить к ним доступ, вам нужно получить доступ к странице через браузер и прочитать ее. Если с момента вашего последнего визита авторы сайта внесли какие-нибудь изменения, местоположение и стиль данных могли измениться.

Вместо того чтобы публиковать веб-страницы, вы можете предоставить доступ к данным через веб-интерфейс программирования приложений (Application Programming Interface, API). Клиенты получают доступ к вашему сервису, делая запросы к URL, и получают ответы, содержащие статус и данные. Вместо HTML-страниц данные имеют формат, который удобнее использовать в других программах вроде JSON и XML (в главе 8 содержится более подробная информация о форматах).

Понятие *«передача состояния представления»* (Representational State Transfer, REST) было определено Роем Филдингом (Roy Fielding) в его докторской диссертации. Многие продукты имеют *REST-интерфейс* или *интерфейс RESTful*. На практике это часто означает, что они имеют веб-интерфейс — определения URL, предназначенные для доступа к веб-сервису.

Служба *RESTful* использует *глаголы* HTTP определенными способами, описанными далее:

- `HEAD` — получает информацию о ресурсе, но не его данные;
- `GET` — как подразумевает имя, `GET` получает данные ресурса с сервера. Это стандартный метод, используемый вашим браузером. В любое время, когда вы видите URL с вопросительным знаком (?), за которым следует несколько аргументов, вы можете распознать запрос `GET`. `GET` не должен использоваться для создания, изменения или удаления данных;
- `POST` — этот глагол обновляет данные на сервере. Он часто используется для HTML-форм и сетевых API;
- `PUT` — этот глагол создает новый ресурс;
- `DELETE` — этот глагол говорит сам за себя: `DELETE` удаляет. Мы за правдивость в рекламе!

Клиент RESTful также может запрашивать содержимое одного или нескольких типов с помощью заголовков запроса HTTP. Например, сложный сервис с интерфейсом REST может принимать и возвращать данные в строках JSON.

JSON

В главе 1 были показаны два фрагмента кода, с помощью которых мы получали информацию о популярных видео на YouTube, а в главе 8 мы узнали о JSON. JSON особенно хорошо подходит для создания веб-серверов и обмена данными. Этот формат наиболее популярен в сетевых API вроде OpenStack.

Поиск и выборка данных

Иногда вам нужно получить немного больше информации — рейтинг фильма, цену акции или доступность продукта, — но информация доступна только на HTML-страницах, при этом она окружена рекламой и посторонним контентом.

Вы можете извлечь необходимую информацию вручную, сделав следующее.

1. Введите URL в браузер.
2. Подождите, пока загрузится удаленная страница.
3. Просмотрите отображенную страницу на предмет необходимой информации.
4. Запишите ее где-нибудь.
5. Повторите процесс для связанных URL.

Однако гораздо более приятно автоматизировать некоторые из этих шагов. Программа, автоматически получающая данные из Сети, называется *краулером* или *веб-пауком* (неприятный термин, если вы арахнофоб). После того как содержимое было получено с одного из удаленных веб-серверов, парсер анализирует ее, чтобы найти иголку в стоге сена.

Если вам нужно мощное решение, объединяющее в себе возможности поиска и выборки данных, вам стоит загрузить Scrapy (http://scrapy.org/):

```
$ pip install scrapy
```

Scrapy — это фреймворк, а не модуль, в отличие от BeautifulSoup. Он имеет больше возможностей, но зачастую его трудно настроить. Чтобы узнать больше о Scrapy, прочтите документацию (http://scrapy.org/) или познакомьтесь с ним по адресу http://bit.ly/using-scrapy.

Получаем HTML-код с помощью BeautifulSoup

Если у вас уже есть HTML-данные с сайта и вы просто хотите извлечь оттуда данные, вам подойдет BeautifulSoup (http://www.crummy.com/software/BeautifulSoup/). Анализировать HTML труднее, чем кажется. Это происходит потому, что большая часть HTML-кода на общедоступных веб-страницах технически некорректна: незакрытые теги, неправильная вложенность и прочие усложнения. Если вы пытаетесь написать свой HTML-анализатор с помощью регулярных выражений, ко-

торые мы рассматривали в главе 7, вы довольно скоро столкнетесь с подобным беспорядком.

Для того чтобы установить BeautifulSoup, введите следующую команду (не забудьте поставить в конце четверку, иначе pip попробует установить более старую версию и, возможно, выдаст ошибку):

```
$ pip install beautifulsoup4
```

Теперь воспользуемся им для того, чтобы получить все ссылки с веб-страницы. Элемент HTML `a` представляет собой ссылку, а `href` — ее атрибут, который представляет собой место назначения ссылки. В следующем примере мы определим функцию `get_links()`, которая делает грязную работу, и основную программу, которая получает один или несколько URL как аргументы командной строки:

```
def get_links(url):
    import requests
    from bs4 import BeautifulSoup as soup
    result = requests.get(url)
    page = result.text
    doc = soup(page)
    links = [element.get('href') for element in doc.find_all('a')]
    return links
if __name__ == '__main__':
    import sys
    for url in sys.argv[1:]:
        print('Links in', url)
        for num, link in enumerate(get_links(url), start=1):
            print(num, link)
        print()
```

Я сохранил эту программу под именем `links.py`, а затем запустил с помощью данной команды:

```
$ python links.py http://boingboing.net
```

Взгляните на первые несколько отображенных строк:

```
Links in http://boingboing.net/
1 http://boingboing.net/suggest.html
2 http://boingboing.net/category/feature/
3 http://boingboing.net/category/review/
4 http://boingboing.net/category/podcasts
5 http://boingboing.net/category/video/
6 http://bbs.boingboing.net/
7 javascript:void(0)
8 http://shop.boingboing.net/
9 http://boingboing.net/about
10 http://boingboing.net/contact
```

Упражнения

1. Если вы еще не установили Flask, сделайте это сейчас. Это также установит werkzeug, jinja2 и, возможно, другие пакеты.
2. Создайте скелет сайта с помощью веб-сервера Flask. Убедитесь, что сервер начинает свою работу по адресу Localhost на стандартном порте 5000. Если ваш компьютер уже использует порт 5000 для чего-то еще, воспользуйтесь другим портом.
3. Добавьте функцию `home()` для обработки запросов к домашней странице. Пусть она возвращает строку `It's alive!`.
4. Создайте шаблон для jinja2, который называется `home.html` и содержит следующий контент:

```
<html>
<head>
<title>It's alive!</title>
<body>
I'm of course referring to {{thing}}, which is {{height}} feet tall and {{color}}.
</body>
</html>
```

5. Модифицируйте функцию `home()` вашего сервера, чтобы она использовала шаблон `home.html`. Передайте ей три параметра для команды `GET`: `thing`, `height` и `color`.

10 Системы

> Есть одна вещь, которую может делать компьютер, но не может большинство людей, — они могут быть запечатаны в картонной коробке и лежать на складе.
>
> *Джек Хэнди*

Каждый день, когда вы используете компьютер, вы выводите в виде списка на экран содержимое каталогов, создаете и удаляете файлы и выполняете другую необходимую работу, даже если это не очень захватывает. Вы также можете выполнить эти (и другие) задачи с помощью программ, написанных на Python. Сможет ли эта сила свести вас с ума или заставить потерять сон? Поживем — увидим.

Python предоставляет множество системных функций, содержащихся в модуле `os` (сокращение от Operating System — операционная система), который мы будем импортировать для большинства программ этой главы.

Файлы

Python, как и многие другие языки, создал свои файловые операции по шаблону Unix. Некоторые функции вроде `chown()` и `chmod()` имеют такие же имена, но при этом появились и некоторые новые функции.

Создаем файл с помощью функции open()

В разделе «Ввод информации в файлы и ее вывод из них» главы 8 вы познакомились с функцией `open()`. В этом разделе также содержалась информация о том, как использовать ее для открытия файла или его создания, если он не существует. Создадим текстовый файл, который называется `oops.txt`:

```
>>> fout = open('oops.txt', 'wt')
>>> print('Oops, I created a file.', file=fout)
>>> fout.close()
```

После этого выполним несколько проверок.

Проверяем существование файла с помощью функции exists()

Для того чтобы убедиться, что файл или каталог действительно существуют, а не являются плодом вашего воображения, можете воспользоваться функцией `exists()`, передав ей относительное или абсолютное имя файла, как показано здесь:

```
>>> import os
>>> os.path.exists('oops.txt')
True
>>> os.path.exists('./oops.txt')
True
>>> os.path.exists('waffles')
False
>>> os.path.exists('.')
True
>>> os.path.exists('..')
True
```

Проверяем тип с помощью функции isfile()

Функции, показанные в этом разделе, проверяют, ссылается ли имя на файл, каталог или символьную ссылку (см. примеры, которые располагаются после описания ссылок).

Первой мы рассмотрим функцию `isfile()`. Она задает простой вопрос: перед нами находится старый добрый законопослушный файл?

```
>>> name = 'oops.txt'
>>> os.path.isfile(name)
True
```

Вот так можно определить папку:

```
>>> os.path.isdir(name)
False
```

Одна точка (`.`) является сокращением для текущей папки, а две точки (`..`) — для родительской. Эти папки существуют всегда, поэтому следующее выражение вернет результат `True`:

```
>>> os.path.isdir('.')
True
```

Модуль `os` содержит множество функций, работающих с *путем к файлу* (полное имя файла, которое начинается с символа / и включает все каталоги). Одна из таких функций, `isabs()`, определяет, является ли аргумент абсолютным путем. Аргумент не обязательно должен быть именем реально существующего файла:

```
>>> os.path.isabs(name)
False
>>> os.path.isabs('/big/fake/name')
True
>>> os.path.isabs('big/fake/name/without/a/leading/slash')
False
```

Копируем файлы с помощью функции copy()

Функция `copy()` находится в другом модуле, `shutil`. В этом примере файл `oops.txt` копируется в файл `ohno.txt`:

```
>>> import shutil
>>> shutil.copy('oops.txt', 'ohno.txt')
```

Функция `shutil.move()` копирует файл, а затем удаляет оригинал.

Изменяем имена файлов с помощью функции rename()

Эта функция соответствует своему названию. В этом примере файл `ohno.txt` переименовывается в `ohwell.txt`:

```
>>> import os
>>> os.rename('ohno.txt', 'ohwell.txt')
```

Создаем ссылки с помощью link() или symlink()

В операционных системах семейства Unix файл существует в одном месте, но может иметь несколько имен, которые называются *ссылками*. Среди низкоуровневых *жестких ссылок* найти все имена заданного файла не так уж легко. *Символьная ссылка* позволяет вам получить одновременно оба имени — оригинальное и новое. Вызов `link()` создает жесткую ссылку, а `symlink()` — символьную ссылку. Функция `islink()` проверяет, является ли файл символьной ссылкой.

Вот так можно создать жесткую ссылку на существующий файл oops.txt из нового файла yikes.txt:

```
>>> os.link('oops.txt', 'yikes.txt')
>>> os.path.isfile('yikes.txt')
True
```

Для того чтобы создать символьную ссылку на существующий файл oops.txt из нового файла jeepers.txt, используйте следующий код:

```
>>> os.path.islink('yikes.txt')
False
>>> os.symlink('oops.txt', 'jeepers.txt')
>>> os.path.islink('jeepers.txt')
True
```

Изменяем разрешения с помощью функции chmod()

В системах Unix функция chmod() изменяет разрешение на использование файла. Можно задать возможность читать, записывать и выполнять файл для пользователя (обычно для вас, если файл создавали вы), основной группы, в которой находится пользователь, и остального мира. Команда принимает сильно сжатое восьмеричное значение (в системе счисления с основанием 8), которое содержит в себе информацию о пользователе, группе и другие разрешения. Например, для того чтобы указать, что файл oops.txt доступен только для чтения своему владельцу, введите следующий код:

```
>>> os.chmod('oops.txt', 0o400)
```

Если вы не хотите работать с таинственными восьмеричными значениями и вам приятнее работать с непонятными (немного) таинственными символами, можете импортировать некоторые константы из модуля stat и использовать выражение, аналогичное следующему:

```
>>> import stat
>>> os.chmod('oops.txt', stat.S_IRUSR)
```

Изменение владельца файла с помощью функции chown()

Эта функция также характерна для систем Unix/Linux/Mac. Вы можете изменить владельца и/или группу, указав числовой идентификатор пользователя ID (uid) и идентификатор группы (gid):

```
>>> uid = 5
>>> gid = 22
>>> os.chown('oops', uid, gid)
```

Получаем pathname с помощью функции abspath()

Эта функция расширяет относительное имя до абсолютного. Если вы находитесь в папке `/usr/gaberlunzie`, в которой лежит файл `oops.txt`, то можете воспользоваться следующим кодом:

```
>>> os.path.abspath('oops.txt')
'/usr/gaberlunzie/oops.txt'
```

Получаем символьную ссылку с помощью функции realpath()

В одном из предыдущих разделов мы создавали символьную ссылку на файл `oops.txt` из нового файла `jeepers.txt`. При похожих обстоятельствах вы можете получить имя файла `oops.txt` из файла `jeepers.txt` с помощью функции `realpath()`, как показано здесь:

```
>>> os.path.realpath('jeepers.txt')
'/usr/gaberlunzie/oops.txt'
```

Удаляем файл с помощью функции remove()

В этом сниппете мы используем функцию `remove()` и попрощаемся с файлом `oops.txt`:

```
>>> os.remove('oops.txt')
>>> os.path.exists('oops.txt')
False
```

Каталоги

В большинстве операционных систем файлы существуют в рамках иерархии каталогов (иначе их еще называют папками). Контейнером для всех этих файлов и каталогов служит *файловая система* (иногда ее называют *томом*). Стандартный модуль `os` работает с такими особенностями и предоставляет функции, с помощью которых вы можете ими манипулировать.

Создаем каталог с помощью функции mkdir()

В этом примере показывается, как создать каталог `poems`, в котором мы сохраним предыдущее стихотворение:

```
>>> os.mkdir('poems')
>>> os.path.exists('poems')
True
```

Удаляем каталог с помощью функции rmdir()

Немного подумав, вы решили, что этот каталог вам не нужен. Удалить его можно вот так:

```
>>> os.rmdir('poems')
>>> os.path.exists('poems')
False
```

Выводим на экран содержимое каталога с помощью функции listdir()

О’кей, дубль два: снова создадим файл poems и что-нибудь в него запишем:

```
>>> os.mkdir('poems')
```

Теперь получим список всех файлов, содержащихся в этом каталоге (которых пока нет):

```
>>> os.listdir('poems')
[]
```

Далее создадим подкаталог:

```
>>> os.mkdir('poems/mcintyre')
>>> os.listdir('poems')
['mcintyre']
```

Создайте в подкаталоге файл (не вводите все эти строки, если только не хотите почувствовать себя поэтом, просто убедитесь, что закрыли все одинарные или тройные кавычки):

```
>>> fout = open('poems/mcintyre/the_good_man', 'wt')
>>> fout.write('''Cheerful and happy was his mood,
... He to the poor was kind and good,
... And he oft' times did find them food,
... Also supplies of coal and wood,
... He never spake a word was rude,
... And cheer'd those did o'er sorrows brood,
... He passed away not understood,
... Because no poet in his lays
... Had penned a sonnet in his praise,
... 'Tis sad, but such is world's ways.
... ''')
344
>>> fout.close()
```

Наконец, проверим, что у нас получилось. Лучше бы ему там быть:

```
>>> os.listdir('poems/mcintyre')
['the_good_man']
```

Изменяем текущий каталог с помощью функции chdir()

С помощью этой функции вы можете переходить из одной папки в другие. Покинем текущую папку и проведем немного времени в каталоге `poems`:

```
>>> import os
>>> os.chdir('poems')
>>> os.listdir('.')
['mcintyre']
```

Перечисляем совпадающие файлы с помощью функции glob()

Функция `glob()` ищет совпадающие имена файлов или каталогов с помощью правил оболочки системы Unix, а не более полного синтаксиса регулярных выражений. Эти правила выглядят так:

- `*` — совпадает со всем (в регулярных выражениях аналогом этого правила является `.*`);
- `?` — совпадает с одним символом;
- `[abc]` — совпадает с символами `a`, `b` или `c`;
- `[!abc]` — совпадает со всеми символами, кроме `a`, `b` или `c`.

Получим все файлы и каталоги, имена которых начинаются с буквы `m`:

```
>>> import glob
>>> glob.glob('m*')
['mcintyre']
```

Как насчет файлов и каталогов с именами, состоящими из двух символов?

```
>>> glob.glob('??')
[]
```

Я думаю о слове из восьми букв, которое начинается с `m` и заканчивается на `e`:

```
>>> glob.glob('m??????e')
['mcintyre']
```

Как насчет чего-то, что начинается с букв `k`, `l` или `m` и заканчивается на букву `e`?

```
>>> glob.glob('[klm]*e')
['mcintyre']
```

Программы и процессы

Когда вы запускаете отдельную программу, ваша операционная система создает один *процесс*. Он использует системные ресурсы (центральный процессор, память, место на диске) и структуры данных в *ядре* операционной системы (файлы и сетевые

соединения, статистика использования и т. д.). Процесс изолирован от других процессов — он не может видеть, что делают другие процессы, или мешать им.

Операционная система отслеживает все запущенные процессы, давая каждому из них немного времени, а затем переключаясь на другие, для того чтобы справедливо распределять работу и реагировать на действия пользователя. Вы можете увидеть состояние своих процессов с помощью графического интерфейса вроде Mac's Activity Monitor (OS X) или Диспетчера задач в Windows.

Вы также можете получать данные о процессах собственных программ. Модуль стандартной библиотеки os помогает получить некоторую системную информацию. Например, следующие функции позволяют получить *идентификатор процесса* и *текущую рабочую папку* запущенного интерпретатора Python:

```
>>> import os
>>> os.getpid()
76051
>>> os.getcwd()
'/Users/williamlubanovic'
```

А эти — мои *идентификаторы пользователя* и *группы*:

```
>>> os.getuid()
501
>>> os.getgid()
20
```

Создаем процесс с помощью модуля subprocess

Все программы, с которыми вы сталкивались до этого момента, представляли собой отдельные процессы. Вы можете запускать и останавливать остальные существующие программы с помощью Python, используя модуль subprocess. Если вы хотите просто запустить другую программу в оболочке и получить результат ее работы (стандартный отчет о работе и отчет об ошибках), используйте функцию getoutput(). В этом примере мы получим результат работы программы date системы Unix:

```
>>> import subprocess
>>> ret = subprocess.getoutput('date')
>>> ret
'Sun Mar 30 22:54:37 CDT 2014'
```

Вы не получите результат, пока процесс не завершится. Если вам нужно вызвать что-то, что может занять много времени, обратитесь к разделу «Конкуренция» главы 11. Поскольку аргументом функции getoutput() является строка, представляющая собой команду оболочки, вы можете включить аргументы, каналы, перенаправление ввода/вывода и т. д.:

```
>>> ret = subprocess.getoutput('date -u')
>>> ret
'Mon Mar 31 03:55:01 UTC 2014'
```

Передача строки-отчета команде wc насчитывает одну строку, шесть «слов» и 29 символов:

```
>>> ret = subprocess.getoutput('date -u | wc')
>>> ret
'       1       6      29'
```

Метод check_output() принимает список команд и аргументов. По умолчанию он возвращает объект типа bytes, а не строки и не использует оболочку:

```
>>> ret = subprocess.check_output(['date', '-u'])
>>> ret
b'Mon Mar 31 04:01:50 UTC 2014\n'
```

Чтобы показать статус выхода другой программы, используйте функцию getstatusoutput(), которая возвращает кортеж, содержащий код статуса и результат работы:

```
>>> ret = subprocess.getstatusoutput('date')
>>> ret
(0, 'Sat Jan 18 21:36:23 CST 2014')
```

Если вам нужен не результат работы программы, а только код, используйте функцию call():

```
>>> ret = subprocess.call('date')
Sat Jan 18 21:33:11 CST 2014
>>> ret
0
```

(В системах семейства Unix 0 обычно является статусом, сигнализирующим об успехе.)

Эти дата и время были выведены на экран, но не получены нашей программой. Поэтому мы сохраняем код возврата как ret.

Вы можете запускать программы с аргументами двумя способами. Первый заключается в том, чтобы разместить их в одной строке. Для примера возьмем команду date -u, которая выводит на экран дату и время в UTC (о UTC мы поговорим немного позже):

```
>>> ret = subprocess.call('date -u', shell=True)
Tue Jan 21 04:40:04 UTC 2014
```

Вам нужно использовать значение shell=True, чтобы распознать команду date -u, разбив ее на отдельные строки и, возможно, расширяя любые символы подстановки вроде * (в нашем примере мы их не использовали).

Во втором варианте мы создаем список аргументов, поэтому нам не нужно вызывать оболочку:

```
>>> ret = subprocess.call(['date', '-u'])
Tue Jan 21 04:41:59 UTC 2014
```

Создаем процесс с помощью модуля multiprocessing

Вы можете запустить функцию Python как отдельный процесс или даже несколько независимых процессов с помощью модуля multiprocessing. Рассмотрим короткий пример, который не делает ничего полезного. Сохраните его под именем mp.py, а затем запустите его, введя команду python mp.py:

```
import multiprocessing
import os
def do_this(what):
    whoami(what)
def whoami(what):
    print("Process %s says: %s" % (os.getpid(), what))
if __name__ == "__main__":
    whoami("I'm the main program")
    for n in range(4):
        p = multiprocessing.Process(target=do_this,
          args=("I'm function %s" % n,))
        p.start()
```

Когда я запускаю этот пример, то вижу на экране следующее:

```
Process 6224 says: I'm the main program
Process 6225 says: I'm function 0
Process 6226 says: I'm function 1
Process 6227 says: I'm function 2
Process 6228 says: I'm function 3
```

Функция Process() породила новый процесс и запустила в нем функцию do_this(). Поскольку мы делали это в цикле с четырьмя итерациями, мы сгенерировали четыре новых процесса, которые выполнили методы do_this() и завершились.

Модуль multiprocessing имеет множество возможностей. Он предназначен для тех случаев, когда вам нужно разбить некую задачу на несколько процессов, чтобы сэкономить время, например загрузить веб-страницу для получения с нее данных, изменить размер изображений и т. д. Он содержит способы разместить задачи в очередь, позволить процессам общаться и подождать, пока все процессы завершатся. В разделе «Конкуренция» главы 11 содержится более подробная информация.

Убиваем процесс с помощью функции terminate()

Если вы создали один или несколько процессов и по какой-то причине хотите завершить один из них (возможно, он застрял в цикле, или вам стало скучно, или вы хотите побыть жестоким правителем), используйте функцию terminate(). В следующем примере наш процесс должен досчитать до миллиона, засыпая после каждо-

го шага на секунду и выводя раздражающее сообщение. Однако у нашей основной программы заканчивается терпение, и она сбивает его с орбиты:

```
import multiprocessing
import time
import os
def whoami(name):
    print("I'm %s, in process %s" % (name, os.getpid()))
def loopy(name):
    whoami(name)
    start = 1
    stop = 1000000
    for num in range(start, stop):
        print("\tNumber %s of %s. Honk!" % (num, stop))
        time.sleep(1)
if __name__ == "__main__":
    whoami("main")
    p = multiprocessing.Process(target=loopy, args=("loopy",))
    p.start()
    time.sleep(5)
    p.terminate()
```

Когда я запускаю эту программу, я вижу следующее:

```
I'm main, in process 97080
I'm loopy, in process 97081
    Number 1 of 1000000. Honk!
    Number 2 of 1000000. Honk!
    Number 3 of 1000000. Honk!
    Number 4 of 1000000. Honk!
    Number 5 of 1000000. Honk!
```

Календари и часы

Программисты прилагают удивительное количество усилий в процессе работы с датами и временем. Поговорим о некоторых проблемах, с которыми они сталкиваются, а затем рассмотрим лучшие способы и приемы, позволяющие проще с ними справиться.

Даты могут быть представлены множеством способов — их даже слишком много. Даже англоговорящие люди, использующие римский календарь, применяют множество вариантов представления простой даты:

- July 29 1984;
- 29 Jul 1984;
- 29/7/1984;
- 7/29/1984.

Помимо других проблем, представление даты может быть двусмысленным. В предыдущих примерах довольно легко определить, что 7 означает месяц, а 29 — день месяца, в основном потому что у месяца не может быть номера 29. Но как насчет даты 1/6/2012? Мы говорим о 6 января или 1 июня?

Название месяца в римском календаре изменяется в зависимости от языка. Даже год и месяц могут иметь разные определения в разных культурах.

Високосные годы — это еще одна проблема. Вы, возможно, знаете, что каждый четвертый год является високосным (в этом году проходят летняя олимпиада и выборы президента в Америке). Знаете ли вы, что каждый сотый год не является високосным, а каждый 400-й — является? Рассмотрим пример кода, в котором проверяется, является ли год високосным:

```
>>> import calendar
>>> calendar.isleap(1900)
False
>>> calendar.isleap(1996)
True
>>> calendar.isleap(1999)
False
>>> calendar.isleap(2000)
True
>>> calendar.isleap(2002)
False
>>> calendar.isleap(2004)
True
```

Работа с временем также может доставить неприятности, особенно из-за часовых поясов и перехода на летнее время. Если вы взглянете на карту часовых поясов, то окажется, что эти пояса больше соответствуют политическим и историческим границам, вместо того чтобы сменяться каждые 15° (360°/24) долготы. Кроме того, разные страны переходят на летнее время и обратно в разные дни года. Фактически страны южного полушария переводят свои часы вперед, когда страны северного полушария переводят их назад, и наоборот (если вы немного задумаетесь, то поймете, почему так происходит).

Стандартная библиотека Python имеет множество модулей для работы с датой и временем: `datetime`, `time`, `calendar`, `dateutil` и др. Их функции немного пересекаются друг с другом, и это может создать путаницу.

Модуль datetime

Начнем с рассмотрения стандартного модуля `datetime`. В нем определены четыре основных объекта, каждый из которых содержит множество методов:

- `date` для годов, месяцев и дней;
- `time` для часов, минут, секунд и долей секунды;

- datetime для даты и времени одновременно;
- timedelta для интервалов даты и/или времени.

Вы можете создать объект date, указав год, месяц и день. Эти значения будут доступны как атрибуты:

```
>>> from datetime import date
>>> halloween = date(2014, 10, 31)
>>> halloween
datetime.date(2014, 10, 31)
>>> halloween.day
31
>>> halloween.month
10
>>> halloween.year
2014
```

Вы можете вывести на экран содержимое объекта date с помощью его метода isoformat():

```
>>> halloween.isoformat()
'2014-10-31'
```

iso в данном контексте ссылается на ISO 8601 — международный стандарт для представления даты и времени. В этом формате мы записываем дату, начиная с самого общего элемента (год) и заканчивая самым точным (день). С его помощью можно также корректно отсортировать даты: сначала по году, затем по месяцу, затем по дню. Я обычно выбираю этот формат для представления данных в программах и для имен файлов, которые сохраняют данные по дате. В следующем разделе будут показаны более сложные методы strptime() и strftime() для анализа и форматирования дат.

В этом примере метод today() используется для генерации сегодняшней даты:

```
>>> from datetime import date
>>> now = date.today()
>>> now
datetime.date(2014, 2, 2)
```

В следующем примере объект timedelta используется для того, чтобы добавить к объекту date некоторый временной интервал:

```
>>> from datetime import timedelta
>>> one_day = timedelta(days=1)
>>> tomorrow = now + one_day
>>> tomorrow
datetime.date(2014, 2, 3)
>>> now + 17*one_day
datetime.date(2014, 2, 19)
>>> yesterday = now - one_day
>>> yesterday
datetime.date(2014, 2, 1)
```

Объект `date` может иметь значение из диапазона, начинающегося с `date.min` (`year=1, month=1, day=1`) и заканчивающегося `date.max` (`year=9999, month=12, day=31`). Вы не можете использовать его для исторических или астрономических расчетов.

Объект `time` модуля `datetime` применяется для представления времени дня:

```
>>> from datetime import time
>>> noon = time(12, 0, 0)
>>> noon
datetime.time(12, 0)
>>> noon.hour
12
>>> noon.minute
0
>>> noon.second
0
>>> noon.microsecond
0
```

Порядок аргументов таков: от самой крупной единицы времени (часа) до самой мелкой (миллисекунды). Если вы передадите не все аргументы, объект `time` предположит, что все они имеют значение `0`. Кстати, несмотря на то, что вы можете сохранять и получать миллисекунды, это не значит, что вы можете получить время вашего компьютера с точностью до миллисекунды. Высокая точность измерений зависит от многих факторов, присущих аппаратному обеспечению и операционной системе.

Объект `datetime` содержит дату и время дня. Вы можете создать такой объект непосредственно, как показано в следующем примере, — мы создадим объект, в который запишем значения «2 января, 2014, 3:04 утра, плюс 5 секунд и 6 миллисекунд»:

```
>>> from datetime import datetime
>>> some_day = datetime(2014, 1, 2, 3, 4, 5, 6)
>>> some_day
datetime.datetime(2014, 1, 2, 3, 4, 5, 6)
```

Объект `datetime` также имеет метод `isoformat()`:

```
>>> some_day.isoformat()
'2014-01-02T03:04:05.000006'
```

Буква `T`, которая находится в середине, разделяет дату и время.

Объект `datetime` имеет метод `now()`, с помощью которого вы можете получить текущие дату и время:

```
>>> from datetime import datetime
>>> now = datetime.now()
>>> now
datetime.datetime(2014, 2, 2, 23, 15, 34, 694988)
14
>>> now.month
2
```

```
>>> now.day
2
>>> now.hour
23
>>> now.minute
15
>>> now.second
34
>>> now.microsecond
694988
```

Вы можете объединить объекты `date` и `time` в объект `datetime` с помощью метода `combine()`:

```
>>> from datetime import datetime, time, date
>>> noon = time(12)
>>> this_day = date.today()
>>> noon_today = datetime.combine(this_day, noon)
>>> noon_today
datetime.datetime(2014, 2, 2, 12, 0)
```

Вы можете получить объекты `date` и `time` из объекта `datetime` с помощью методов `date()` и `time()`:

```
>>> noon_today.date()
datetime.date(2014, 2, 2)
>>> noon_today.time()
datetime.time(12, 0)
```

Модуль time

В Python имеется модуль `datetime`, имеющий объект `time`, а также отдельный модуль `time`, что создает путаницу. Дальше — больше, в модуле `time` имеется функция с именем — что вы подумали? — `time()`.

Одним из способов представления абсолютного времени является подсчет количества секунд, прошедших с некоторой стартовой точки. В Unix используется количество секунд, прошедших с полуночи 1 января 1970 года (примерно в это время появилась система Unix). Это значение часто называют `epoch`, и зачастую оно является простейшим способом обмениваться датой и временем между системами.

Функция `time()` модуля `time` возвращает текущее время как значение `epoch`:

```
>>> import time
>>> now = time.time()
>>> now
1391488263.664645
```

Если выполнить подсчеты, вы увидите, что прошло более миллиарда секунд после наступления нового, 1970 года. И куда ушло время?

Вы можете преобразовать значение epoch в строку с помощью функции ctime():

```
>>> time.ctime(now)
'Mon Feb  3 22:31:03 2014'
```

В следующем разделе вы увидите, как создавать более приятные глазу форматы для даты и времени.

Значения epoch полезны для обмена датой и временем с разными системами вроде JavaScript. Однако иногда вам нужно получить именно значения дней, часов и т. д., объект time предоставляет их как объекты struct_time. Функция localtime() предоставляет время в вашем текущем часовом поясе, а функция gmtime() — в UTC:

```
>>> time.localtime(now)
time.struct_time(tm_year=2014, tm_mon=2, tm_mday=3, tm_hour=22, tm_min=31,
tm_sec=3, tm_wday=0, tm_yday=34, tm_isdst=0)
>>> time.gmtime(now)
time.struct_time(tm_year=2014, tm_mon=2, tm_mday=4, tm_hour=4, tm_min=31,
tm_sec=3, tm_wday=1, tm_yday=35, tm_isdst=0)
```

В моем (Центральном) часовом поясе 22:31 — это то же самое, что 04:31 следующего дня в поясе UTC (раньше его называли Гринвичским временем или временем Зулу). Если вы опустите аргумент функции localtime() или gmtime(), они предположат, что сконвертировать нужно текущее время.

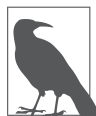
Их противоположностью является функция mktime(), которая преобразует объект struct_time в секунды epoch:

```
>>> tm = time.localtime(now)
>>> time.mktime(tm)
1391488263.0
```

Результат не совсем похож на предыдущее значение epoch, полученное с помощью функции now(), поскольку объект struct_time сохраняет время лишь до секунд.

Небольшой совет: везде, где возможно, используйте часовой пояс UTC. UTC — это абсолютное время, не зависящее от часовых поясов. Если у вас есть сервер, установите его время согласно часовому поясу UTC, не используйте местное время.

Еще один совет (совершенно бесплатный): **никогда не используйте летнее время**, если можете этого избежать. Если вы используете летнее время, весной один час выпадет из календаря, а осенью «удвоится». По каким-то причинам многие организации пользуются летним временем в своих компьютерных системах, а потом удивляются удвоению и потере данных. Заканчивается все печально.



Не забывайте: UTC для времени, UTF-8 для строк (о UTF-8 подробнее можно прочитать в главе 7).

Читаем и записываем дату и время

Функция Isoformat() — это не единственный способ записывать дату и время. Вы уже видели функцию ctime() в модуле time, которую можете использовать для преобразования времени epoch в строку:

```
>>> import time
>>> now = time.time()
>>> time.ctime(now)
'Mon Feb  3 21:14:36 2014'
```

Вы также можете преобразовывать дату и время с помощью функции strftime(). Она предоставляется как метод в объектах datetime, date и time objects и как функция в модуле time. strftime() использует для вывода информации на экран спецификаторы формата, которые вы можете увидеть в табл. 10.1.

Таблица 10.1. Спецификаторы вывода для strftime()

Спецификатор	**Единица даты/времени**	**Диапазон**
%Y	Год	1900–...
%m	Месяц	01–12
%B	Название месяца	Январь, ...
%b	Сокращение для месяца	Янв, ...
%d	День месяца	01–31
%A	Название дня	Воскресенье, ...
a	Сокращение для дня	Вск, ...
%H	Часы (24 часа)	00–23
%I	Часы (12 часов)	01–12
%p	AM или PM	AM, PM
%M	Минуты	00–59
%S	Секунды	00–59

К числам слева добавляется ноль.

Рассмотрим пример работы функции strftime(), предоставленной модулем time. Она преобразует объект struct_time в строку. Сначала мы определим строку формата fmt и будем использовать ее снова в дальнейшем:

```
>>> import time
>>> fmt = "It's %A, %B %d, %Y, local time %I:%M:%S%p"
>>> t = time.localtime()
>>> t
time.struct_time(tm_year=2014, tm_mon=2, tm_mday=4, tm_hour=19,
```

```
tm_min=28, tm_sec=38, tm_wday=1, tm_yday=35, tm_isdst=0)
>>> time.strftime(fmt, t)
"It's Tuesday, February 04, 2014, local time 07:28:38PM"
```

Если мы попробуем сделать это с объектом `date`, функция отработает только для даты, время будет установлено в полночь:

```
>>> from datetime import date
>>> some_day = date(2014, 7, 4)
>>> fmt = "It's %B %d, %Y, local time %I:%M:%S%p"
>>> some_day.strftime(fmt)
"It's Friday, July 04, 2014, local time 12:00:00AM"
```

Для объекта `time` будут преобразованы только части, касающиеся времени:

```
>>> from datetime import time
>>> some_time = time(10, 35)
>>> some_time.strftime(fmt)
"It's Monday, January 01, 1900, local time 10:35:00AM"
```

Очевидно, вам не нужно использовать те части объекта `time`, которые касаются дней, поскольку они бессмысленны.

Чтобы пойти другим путем и преобразовать строку к дате или времени, используйте функцию `strptime()` с такой же строкой формата. Эта строка работает не так, как регулярные выражения, — части строки, не касающиеся формата (без символа `%`), должны совпадать точно. Укажем формат «год-месяц-день» вроде `2012-01-29`. Что произойдет, если строка даты, которую вы хотите проанализировать, имеет пробелы вместо дефисов?

```
>>> import time
>>> fmt = "%Y-%m-%d"
>>> time.strptime("2012 01 29", fmt)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/Library/Frameworks/Python.framework/Versions/3.3/lib/
  python3.3/_strptime.py", line 494, in _strptime_time
    tt = _strptime(data_string, format)[0]
  File "/Library/Frameworks/Python.framework/Versions/3.3/lib/
  python3.3/_strptime.py", line 337, in _strptime
    (data_string, format))
ValueError: time data '2012 01 29' does not match format '%Y-%m-%d'
```

Будет ли довольна функция `strptime()`, если мы передадим ей несколько дефисов?

```
>>> time.strptime("2012-01-29", fmt)
time.struct_time(tm_year=2012, tm_mon=1, tm_mday=29, tm_hour=0, tm_min=0,
tm_sec=0, tm_wday=6, tm_yday=29, tm_isdst=-1)
```

Да.

Даже если строка совпадает с заданным форматом, будет сгенерировано исключение, если одно из значений находится вне диапазона:

```
>>> time.strptime("2012-13-29", fmt)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/Library/Frameworks/Python.framework/Versions/3.3/lib/
  python3.3/_strptime.py", line 494, in _strptime_time
    tt = _strptime(data_string, format)[0]
  File "/Library/Frameworks/Python.framework/Versions/3.3/lib/
  python3.3/_strptime.py", line 337, in _strptime
    (data_string, format))
ValueError: time data '2012-13-29' does not match format '%Y-%m-%d'
```

Имена соответствуют вашей *локали* — набору настроек операционной системы для интернационализации. Чтобы вывести на экран другие названия месяцев и дней, измените свою локаль с помощью функции `setlocale()`: ее первый аргумент должен быть равен `locale.LC_TIME` для даты и времени, а второй аргумент — это строка, содержащая сокращение языка и страны. Пригласим на нашу вечеринку в честь Дня всех святых наших иностранных друзей. Мы выведем на экран дату (месяц, число и день недели) на английском, французском, немецком, испанском и исландском. (А что? Думаете, исландцы не любят вечеринки? У них даже есть настоящие эльфы.)

```
>>> import locale
>>> from datetime import date
>>> halloween = date(2014, 10, 31)
>>> for lang_country in ['en_us', 'fr_fr', 'de_de', 'es_es', 'is_is',]:
...     locale.setlocale(locale.LC_TIME, lang_country)
...     halloween.strftime('%A, %B %d')
...
'en_us'
'Friday, October 31'
'fr_fr'
'Vendredi, octobre 31'
'de_de'
'Freitag, Oktober 31'
'es_es'
'viernes, octubre 31'
'is_is'
'föstudagur, október 31'
>>>
```

Откуда можно взять эти волшебные значения аргумента `lang_country`? Это немного ненадежно, но вы можете получить их все сразу (всего их несколько сотен):

```
>>> import locale
>>> names = locale.locale_alias.keys()
```

Из переменной names получим только те имена локалей, которые будут работать с методом setlocale(), вроде тех, что мы использовали в предыдущем примере, — двухсимвольный код языка (http://bit.ly/iso-639-1), в котором после подчеркивания следует двухсимвольный код страны (http://bit.ly/iso-3166-1):

```
>>> good_names = [name for name in names if \
len(name) == 5 and name[2] == '_']
```

Как будут выглядеть первые пять из них?

```
>>> good_names[:5]
['sr_cs', 'de_at', 'nl_nl', 'es_ni', 'sp_yu']
```

Если вы хотите получить все локали для Германии, используйте следующий код:

```
>>> de = [name for name in good_names if name.startswith('de')]
>>> de
['de_at', 'de_de', 'de_ch', 'de_lu', 'de_be']
```

Альтернативные модули

Если вы считаете, что модули стандартной библиотеки только создают путаницу или им не хватает некоторого определенного преобразования, вы можете использовать альтернативные модули от сторонних разработчиков. Рассмотрим несколько из них.

- arrow (http://crsmithdev.com/arrow/). Этот модуль содержит множество функций для работы с датой и временем и имеет простой API.
- dateutil (http://labix.org/python-dateutil). Модуль может проанализировать любой формат даты и хорошо работает с относительными датами и временем.
- iso8601 (https://pypi.python.org/pypi/iso8601). Этот модуль заполняет пробелы, связанные с работой модулей стандартной библиотеки, когда речь идет о формате ISO 8601.
- fleming (https://github.com/ambitioninc/fleming). Модуль содержит множество функций для работы с часовыми поясами.

Упражнения

1. Запишите текущие дату и время как строку в текстовый файл today.txt.
2. Прочтите текстовый файл today.txt и разместите данные в строке today_string.
3. Разберите дату из строки today_string.
4. Выведите на экран список файлов текущего каталога.

5. Выведите на экран список файлов родительского каталога.
6. Используйте модуль multiprocessing, чтобы создать три отдельных процесса. Заставьте каждый из них ждать случайное количество секунд (от одной до пяти), вывести текущее время и завершить работу.
7. Создайте объект `date`, содержащий дату вашего рождения.
8. В какой день недели вы родились?
9. Когда вам будет (или уже было) 10 000 дней от роду?

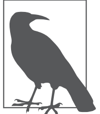
11 Конкуренция и сети

> Время — такая штука, при помощи которой природа не позволяет всем событиям произойти сразу. Пространства — такая штука, благодаря которой все это не происходит со мной.
>
> *Альберт Эйнштейн*

До этого момента большинство программ, которые вы писали, запускались в одном месте (на одном компьютере) по одной строке за раз (последовательные). Но мы можем делать больше одного дела одновременно (конкуренция) и в нескольких местах сразу (распределенные вычисления, или работа с сетями). Существует несколько хороших причин бросить вызов пространству и времени.

- Производительность. Ваша цель заключается в том, чтобы более быстрые компоненты были постоянно заняты, а не ждали более медленных.
- Прочность. Один в поле не воин, поэтому вы хотите продублировать задачи, чтобы обойти недостатки аппаратной и программной частей.
- Простота. Хорошим тоном является разбиение сложных задач на много простых, которые проще создать, понять и исправить.
- Коммуникация. Отправлять независимые байты куда-нибудь далеко, чтобы они пришли с друзьями, очень весело.

Мы начнем с рассмотрения конкуренции, основываясь поначалу на несетевых приемах, описанных в главе 10, — процессах и потоках. Далее рассмотрим остальные подходы вроде функций обратного вызова, зеленых потоков и сопрограмм. Наконец, поговорим о работе с сетями, изначально в рамках вопроса о конкуренции, а затем и целиком.



Некоторые пакеты Python, рассмотренные в этой главе, еще не были портированы в Python 3 на момент написания книги. В большинстве случаев я буду показывать вам код, который нужно запускать с помощью интерактивного интерпретатора Python 2, который мы называем python2.

Конкуренция

Официальный сайт Python рассматривает тему конкуренции в общих чертах и с точки зрения стандартной библиотеки (http://bit.ly/concur-lib). Эти страницы содержат множество ссылок на различные пакеты и приемы, мы покажем наиболее полезные из них в этой главе.

Когда речь идет о компьютерах, вам приходится ждать чего-то по одной из двух причин:

- ограничения ввода-вывода. Эта причина распространена шире других. Процессоры компьютеров безумно быстры — в сотни раз быстрее, чем компьютерная память, и в тысячи — чем диски или сети.
- ограничения процессора. Это случается, если выполняется большое количество объемных задач наподобие научных или графических расчетов.

С конкуренцией связаны еще два термина:

- синхронность — одна вещь следует за другой, как на похоронной процессии;
- асинхронность — задачи независимы, как кошки, которые гуляют сами по себе.

По мере продвижения от простых систем и задач к проблемам реальной жизни в какой-то момент вам придется решить проблему конкуренции. Например, рассмотрим сайт. Вы, как правило, можете предоставить статическую и динамическую страницы довольно быстро. Если ожидание длится долю секунды, приложение считается интерактивным, но если время до отображения или взаимодействия более продолжительное, люди становятся нетерпеливыми. Тесты, проведенные компаниями Google и Amazon, показали, что трафик быстро падает, если страница загружается хоть немного медленнее обычного.

Но что, если вы не можете повлиять на то, что долго выполняется, например загрузка файла на сервер, изменение размеров изображения или запрос к базе данных? Вы больше не можете делать это с помощью синхронного кода, поскольку кто-то уже ждет.

Если вы хотите выполнить несколько задач как можно быстрее на одном компьютере, вы можете сделать их независимыми. Медленные задачи не будут блокировать остальные.

В разделе «Программы и процессы» главы 10 показано, как многопроцессорная обработка может быть использована для того, чтобы распараллелить работу на одной машине. Если вам нужно изменить размер изображения, ваш веб-сервер может создать отдельный процесс, посвященный именно этой задаче, и запустить его асинхронно. Можно масштабировать приложение горизонтально, вызвав несколько процессов изменения размера.

Идея заключается в том, чтобы заставить их работать друг с другом. Наличие любого общего элемента управления или состояния означает, что будут возникать узкие места. Обрабатывать ошибки еще сложнее, поскольку конкурентные

вычисления труднее, чем обычные. Многое может пойти не так, и ваши шансы на успех меньше обычных.

Какие же методы могут помочь вам справиться с этими сложностями? Начнем с хорошего способа, который помогает справиться с несколькими задачами, — очереди.

Очереди

Очередь похожа на список: элементы добавляются с одного ее конца и выходят с другого. Часто такой принцип называют FIFO (first in, first out — «первым пришел — первым ушел»).

Представьте, что вы моете посуду. Если вы делаете работу целиком, вам нужно вымыть каждую тарелку, высушить ее и отложить в сторону. Вы можете сделать это несколькими способами. Можете вымыть первую тарелку, высушить ее и отложить в сторону, а затем повторить для второй и последующих тарелок. Или же можете сгруппировать операции и сначала помыть всю посуду, затем высушить ее целиком, а затем отложить ее в сторону, при этом подразумевается, что в раковине и сушилке достаточно места, чтобы разместить там всю посуду на каждом шаге. Все эти подходы являются синхронными — один работник выполняет одно действие в любой момент времени.

В качестве альтернативы вы могли бы найти одного-двух помощников. Если вы мойщик, то можете вручать каждую вымытую тарелку сушильщику, который будет вручать каждую высохшую тарелку тому, кто отложит ее в сторону. Если все работают в одном темпе, вы должны закончить работу гораздо быстрее, чем если бы делали ее целиком самостоятельно.

Но что, если вы моете посуду быстрее, чем сушильщик успевает с ней справляться? Либо влажная посуда падает на пол, либо вы будете складывать ее между собой и сушильщиком, либо вы просто что-нибудь насвистываете до тех пор, пока сушильщик не будет готов. А если последний человек медленнее сушильщика, сухая посуда будет либо падать на пол, либо накапливаться или насвистывать начнет уже сушильщик. У вас есть несколько работников, но общая задача все еще синхронна и может выполняться только со скоростью самого медленного работника.

«Берись дружно, не будет грузно» — гласит старая пословица (я всегда думал, что это пословица амишей, поскольку она заставляет меня думать о строительстве сарая). Добавление работников может помочь построить сарай или вымыть посуду быстрее. При этом будут задействованы очереди.

В общем случае очереди переносят сообщения, которые могут содержать любую информацию. В нашем случае мы заинтересованы в создании очереди для распределенного управления задачами, также известной как очередь заданий. Каждая тарелка в раковине выдается доступному мойщику, который моет ее и отдает сушильщику, который сушит ее и отдает человеку, убирающему ее в сторону. Этот

процесс может быть синхронным (работники ждут, когда им дадут тарелку, а затем ждут, когда освободится следующий в очереди работник) или асинхронным (посуда поступает от работников с разной скоростью). Если у вас есть достаточно работников и они трудятся в одном темпе, задача будет выполнена гораздо быстрее.

Процессы

Очереди вы можете реализовать множеством способов. Для одного компьютера модуль стандартной библиотеки multiprocessing (с которым вы можете познакомиться в разделе «Программы и процессы» главы 10) содержит функцию `Queue`. Симулируем процессы одного мойщика посуды и одного сушильщика (кто-то может отложить посуду в сторону позже), а также промежуточную очередь `dish_queue`. Назовите эту программу `dishes.py`:

```
import multiprocessing as mp
def washer(dishes, output):
    for dish in dishes:
        print('Washing', dish, 'dish')
        output.put(dish)
def dryer(input):
    while True:
        dish = input.get()
        print('Drying', dish, 'dish')
        input.task_done()
dish_queue = mp.JoinableQueue()
dryer_proc = mp.Process(target=dryer, args=(dish_queue,))
dryer_proc.daemon = True
dryer_proc.start()
dishes = ['salad', 'bread', 'entree', 'dessert']
washer(dishes, dish_queue)
dish_queue.join()
```

Запустите новую программу:

```
$ python dishes.py
Washing salad dish
Washing bread dish
Washing entree dish
Washing dessert dish
Drying salad dish
Drying bread dish
Drying entree dish
Drying dessert dish
```

Эта очередь похожа на простой итератор, который создает набор тарелок. В действительности здесь создаются отдельные процессы, общающиеся между собой. Я использовал `JoinableQueue` и последний метод `join()`, чтобы дать знать мойщику,

что вся посуда была высушена. В модуле multiprocessing существуют очереди и других типов, вы можете обратиться к документации, чтобы получить больше примеров.

Потоки

Поток работает внутри процесса, имея доступ ко всему, что находится в процессе, — это похоже на раздвоение личности. Модуль multiprocessing имеет кузена по имени threading, который использует потоки вместо процессов (на самом деле модуль multiprocessing был разработан позже своего собрата, основанного на процессах). Переделаем наш пример с процессами для использования потоков:

```
import threading
def do_this(what):
    whoami(what)
def whoami(what):
    print("Thread %s says: %s" % (threading.current_thread(), what))
if __name__ == "__main__":
    whoami("I'm the main program")
    for n in range(4):
        p = threading.Thread(target=do_this,
          args=("I'm function %s" % n,))
        p.start()
```

Вот что я вижу на своем экране:

```
Thread <_MainThread(MainThread, started 140735207346960)> says: I'm the main
program
Thread <Thread(Thread-1, started 4326629376)> says: I'm function 0
Thread <Thread(Thread-2, started 4342157312)> says: I'm function 1
Thread <Thread(Thread-3, started 4347412480)> says: I'm function 2
Thread <Thread(Thread-4, started 4342157312)> says: I'm function 3
```

Мы можем воссоздать пример о посуде, основанный на процессах, с помощью потоков:

```
import threading, queue
import time
def washer(dishes, dish_queue):
    for dish in dishes:
    print ("Washing", dish)
        time.sleep(5)
        dish_queue.put(dish)
def dryer(dish_queue):
    while True:
        dish = dish_queue.get()
        print ("Drying", dish)
        time.sleep(10)
```

```
        dish_queue.task_done()
dish_queue = queue.Queue()
for n in range(2):
    dryer_thread = threading.Thread(target=dryer, args=(dish_queue,))
        dryer_thread.start()
dishes = ['salad', 'bread', 'entree', 'desert']
washer(dishes, dish_queue)
dish_queue.join()
```

Различие между модулями multiprocessing и threading заключается в том, что модуль threading не имеет функции `terminate()`. Не существует простого способа завершить запущенный поток, поскольку это может вызвать разнообразные проблемы в коде и, возможно, даже в пространственно-временном континууме.

Потоки могут быть опасны. Как и управление памятью вручную в языках вроде C и C++, они могут вызвать появление ошибок, которые ужасно трудно найти и исправить. Для того чтобы использовать потоки, весь код программы — и код внешних библиотек, которые он использует, — должен быть потокобезопасным. В предыдущем примере кода потоки не работали с глобальными переменными, поэтому они могли работать независимо, ничего не разрушая.

Представьте, что вы исследуете паранормальную активность в доме с привидениями. Привидения скитаются по коридорам, но ни одно из них не знает о другом, и в любой момент любое из них может просмотреть, добавить, удалить или переместить домашнюю обстановку.

Вы настороженно идете по дому, снимая показатели со своих впечатляющих инструментов. И внезапно замечаете, что подсвечник, мимо которого прошли несколько секунд назад, пропал.

Содержимое дома похоже на переменные программы. Привидения — это потоки процесса (дома). Если бы привидения только просматривали содержимое дома, проблемы бы не было — поток просто считает значение константы или переменной, не пытаясь его изменить.

Однако некая невидимая сущность может схватить ваш фонарик, дунуть холодной струей воздуха на вашу шею, рассыпать шарики на ступеньках или заставить вспыхнуть огонь в камине. Особо утонченные привидения изменили бы что-нибудь в другой комнате, чего вы бы даже не заметили.

Несмотря на ваши шикарные инструменты, вам будет очень трудно разобраться в том, кто, как и когда это сделал.

Если бы вы использовали несколько процессов вместо потоков, это было бы похоже на несколько домов, в которых обитает только одно (живое) существо. Если бы вы поставили бутылку бренди перед камином, она все еще была бы на своем месте час спустя. Возможно, немного жидкости испарилось бы, но сама бутылка осталась бы на том же месте.

Потоки могут быть полезны и безопасны, когда речь не идет о глобальных данных. В частности, потоки полезно использовать для экономии времени при ожидании завершения некой операции ввода/вывода. В этих случаях потокам

не придется сражаться за данные, поскольку у каждого из них имеется свой набор переменных.

Но потоки иногда могут менять глобальные данные по хорошей причине. Фактически самая распространенная причина использования нескольких потоков — это возможность разделить между ними работу над некоторыми данными, поэтому можно ожидать, что некоторые данные будут изменены.

Классический способ разделить данные безопасно — разместить программную блокировку перед изменением переменной в потоке. Это позволит оградить ее значение от других потоков и внести свои изменения. В примере с домом вы бы просто оставили бригаду охотников за привидениями в той комнате, которая должна остаться свободной от привидений. Вам лишь нужно не забывать разблокировать ее. Блокировки также могут быть вложенными — что, если другая бригада охотников за привидениями также будет наблюдать за этой же комнатой или за всем домом? Использование блокировок является традицией, но печально известно тем, что его трудно организовать правильно.

В Python потоки не ускоряют задачи, связанные с ограничениями процессора, из-за одной детали реализации стандартной системы Python, которая называется Global Interpreter Lock (GIL). Она предназначена для того, чтобы избежать потоковых проблем в интерпретаторе Python, и действительно может замедлить многопоточную программу по сравнению с однопоточной или даже многопроцессорной версией.

Рассмотрим рекомендации для работы с Python.

- Используйте потоки для задач, связанных с ограничениями ввода-вывода.
- Используйте процессы, сетевые вычисления или события (которые мы рассмотрим в следующем разделе) для задач, связанных с ограничениями процессора.

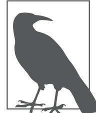
Зеленые потоки и gevent

Как вы уже видели, разработчики стремятся избежать медленных мест в программах, запуская их в отдельных потоках или процессах. Примером такого дизайна является веб-сервер Apache.

Альтернативой этому подходу является программирование, основанное на событиях. Программа, основанная на событиях, запускает центральный цикл обработки событий, раздает задачи и повторяет цикл. Так устроен веб-сервер nginx, он работает быстрее Apache.

Библиотека gevent основана на событиях и позволяет достичь следующего: вы пишете обычный императивный код, и он волшебным образом превращается в сопрограммы. Они похожи на генераторы, которые могут взаимодействовать друг с другом и отслеживать свое текущее состояние. Библиотека gevent модифицирует многие стандартные объекты Python вроде `socket` для того, чтобы использовать

его механизм вместо блокирования. Это не сработает для кода надстроек Python, который написан на C, например для некоторых драйверов баз данных.



На момент написания этой книги библиотека gevent не была полностью портирована на Python 3, поэтому примеры кода используют инструменты Python 2 pip2 и python2.

Вы можете установить библиотеку gevent с помощью версии pip для Python 2:

```
$ pip2 install gevent
```

Так выглядит пример кода на сайте библиотеки gevent (http://www.gevent.org/). Вы увидите функцию `gethostbyname()` класса `socket` в следующем разделе DNS. Эта функция работает синхронно, поэтому вам придется подождать (возможно, много секунд), пока она не получит имена серверов со всего мира, чтобы найти нужный адрес. Но вы можете использовать версию gevent, чтобы искать несколько сайтов независимо друг от друга. Сохраните этот файл как `gevent_test.py`:

```
import gevent
from gevent import socket
hosts = ['www.crappytaxidermy.com', 'www.walterpottertaxidermy.com',
    'www.antique-taxidermy.com']
jobs = [gevent.spawn(gevent.socket.gethostbyname, host) for host in hosts]
gevent.joinall(jobs, timeout=5)
for job in jobs:
    print(job.value)
```

В этом примере вы можете увидеть однострочный цикл `for`. Каждое имя хоста по очереди передается в вызов `gethostbyname()`, но они могут быть запущены асинхронно благодаря версии функции `gethostbyname()` библиотеки gevent.

Запустите файл `gevent_test.py` с помощью Python 2, введя следующее:

```
$ python2 gevent_test.py
66.6.44.4
74.125.142.121
78.136.12.50
```

Функция `gevent.spawn()` создает зеленый поток (его также иногда называют микропотоком) для выполнения каждого вызова `gevent.socket.gethostbyname(url)`.

Разница между ним и обычным потоком заключается в том, что зеленый поток не блокируется. Если произошло какое-то событие, которое заблокировало бы обычный поток, gevent переключит управление на другой зеленый поток.

Метод `gevent.joinall()` ожидает завершения всех созданных задач. Наконец, мы выводим на экран IP-адреса, полученные для заданных имен хостов.

Вместо класса `socket` модуля gevent вы можете использовать его функции для monkey-patching (обезьяний патч). Они модифицируют стандартные модули вроде `socket` так, чтобы они использовали зеленые потоки вместо того, чтобы каждый

раз вызывать версию модуля gevent. Это полезно, если вы хотите использовать gevent везде, даже в коде, к которому вы можете не иметь доступа.

Добавьте в начало программы следующий вызов:

```
from gevent import monkey
monkey.patch_socket()
```

Это заменит все обычные сокеты на сокеты gevent даже в стандартной библиотеке. Но это работает только для кода Python, но не для библиотек, написанных на C.

Еще одна функция выполняет такой патчинг для еще большего количества модулей стандартной библиотеки:

```
from gevent import monkey
monkey.patch_all()
```

Разместите этот код в начале программы, чтобы максимально воспользоваться ускорением, обеспечиваемым gevent.

Сохраните программу под именем `gevent_monkey.py`:

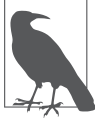
```
import gevent
from gevent import monkey; monkey.patch_all()
import socket
hosts = ['www.crappytaxidermy.com', 'www.walterpottertaxidermy.com',
    'www.antique-taxidermy.com']
jobs = [gevent.spawn(socket.gethostbyname, host) for host in hosts]
gevent.joinall(jobs, timeout=5)
for job in jobs:
    print(job.value)
```

Запустите программу с помощью Python 2:

```
$ python2 gevent_monkey.py
66.6.44.4
74.125.192.121
78.136.12.50
```

Использование gevent может нести потенциальную опасность. Как и в случае с любой другой системой, основанной на событиях, каждый исполняемый вами фрагмент кода должен быть относительно быстрым. Несмотря на то что код, который выполняет много работы, не блокируется, он будет работать медленно.

Сама идея monkey-patching заставляет нервничать некоторых людей. Несмотря на это, многие крупные сайты вроде Pinterest используют gevent для значительного ускорения своей работы. Используйте gevent строго по назначению, как таблетки по рецепту.



Существуют два других популярных фреймворка, основанных на событиях, — tornado (http://www.tornadoweb.org/) и gunicorn (http://gunicorn.org/). Они помогают обрабатывать события на низком уровне, а также предоставляют быстрый веб-сервер. Их стоит рассмотреть, если вы хотите создать быстрый сайт, не применяя традиционные веб-серверы вроде Apache.

twisted

twisted (http://twistedmatrix.com/trac/) — это асинхронный фреймворк для работы с сетями, управляемый событиями. Вы подключаете функции к событиям вроде получения данных или закрытия соединения, и эти функции вызываются, когда событие случается. Эти функции называются функциями обратного вызова, и если вы уже писали код на языке JavaScript, он может показаться вам знакомым. Если же он для вас в новинку, то может показаться вывернутым наизнанку. Некоторым разработчикам может оказаться труднее поддерживать код, основанный на функциях обратного вызова, по мере роста приложения.

Как и gevent, twisted еще не был портирован на Python 3. В этом разделе мы будем использовать установщик и интерактивный интерпретатор Python 2. Чтобы установить фреймворк, введите следующую команду:

```
$ pip2 install twisted
```

twisted — это крупный пакет, который поддерживает множество интернет-протоколов на базе TCP и UDP. Для краткости мы рассмотрим небольшой сервер и клиент, созданные на базе примеров для twisted (http://bit.ly/twisted-ex). Сначала рассмотрим сервер, `knock_server.py` (обратите внимание на синтаксис Python 2 для функции `print()`):

```
from twisted.internet import protocol, reactor
class Knock(protocol.Protocol):
    def dataReceived(self, data):
        print 'Client:', data
        if data.startswith("Knock knock"):
            response = "Who's there?"
        else:
            response = data + " who?"
        print 'Server:', response
        self.transport.write(response)
class KnockFactory(protocol.Factory):
    def buildProtocol(self, addr):
        return Knock()
reactor.listenTCP(8000, KnockFactory())
reactor.run()
```

Теперь взглянем на его верного компаньона, `knock_client.py`:

```
from twisted.internet import reactor, protocol
class KnockClient(protocol.Protocol):
    def connectionMade(self):
        self.transport.write("Knock knock")
    def dataReceived(self, data):
        if data.startswith("Who's there?"):
            response = "Disappearing client"
```

```
            self.transport.write(response)
        else:
            self.transport.loseConnection()
            reactor.stop()
class KnockFactory(protocol.ClientFactory):
    protocol = KnockClient
def main():
    f = KnockFactory()
    reactor.connectTCP("localhost", 8000, f)
    reactor.run()
if __name__ == '__main__':
    main()
```

Сначала запустим сервер:

```
$ python2 knock_server.py
```

Потом — клиент:

```
$ python2 knock_client.py
```

Сервер и клиент обмениваются сообщениями, и сервер выводит весь диалог:

```
Client: Knock knock
Server: Who's there?
Client: Disappearing client
Server: Disappearing client who?
```

Наш клиент-шутник завершает работу, оставляя сервер ждать ударной реплики.

Если вы хотите забраться в дебри, попробуйте запустить другие примеры из его документации.

asyncio

Недавно Гвидо ван Россум (помните его?) начал заниматься проблемой конкуренции в Python. У многих пакетов был собственный цикл событий, и каждый из них хотел быть единственным. Как могут примириться механизмы вроде функций обратного вызова, зеленых потоков и др.? После продолжительных дискуссий он предложил решение: модуль `asyncio` (от Asynchronous IO Support Rebooted — асинхронная поддержка ввода-вывода, http://bit.ly/pep-3156) под кодовым именем `Tulip`. Он появился в Python 3.4 под именем `asyncio`. Теперь же он предлагает цикл событий, который совместим с twisted, gevent и другими асинхронными методами. Цель его создания заключается в том, чтобы предоставить стандартный, чистый, отлаженный асинхронный API. Вы сможете наблюдать за тем, как он расширяется, в будущих релизах Python.

Redis

Приведенные раньше примеры кода о мытье посуды, где использовались процессы или потоки, запускались на одной машине. Рассмотрим еще один подход к реализации очередей, которые могут запускаться на одной машине или во всей сети. Даже несмотря на наличие нескольких процессов и/или потоков, иногда одной машины недостаточно. Вы можете считать этот раздел мостиком между конкуренцией на одной машине и конкуренцией на нескольких машинах.

Чтобы запустить примеры из этого раздела, вам нужен сервер Redis и его модуль для Python. Чтобы узнать, как их скачать, обратитесь к разделу «Redis» главы 8. В этой главе Redis используется как база данных. Здесь же мы рассмотрим ту его грань, которая работает с конкуренцией.

Создать очередь можно с помощью списка Redis. Сервер Redis работает на одной машине, на которой могут быть запущены и клиенты. Возможно также, что никакие клиенты на ней не запускаются, а остальные машины получают доступ к серверу по сети. В любом случае клиент общается с сервером с помощью протокола TCP. Один или несколько клиентов-провайдеров помещают сообщения в конец списка. Один или несколько клиентов-работников наблюдают за списком и используют операцию «блокирующее выталкивание». Если список пуст, то все они просто проводят время впустую. Как только появляется сообщение, его получает первый желающий работник.

Как и в предыдущем примере, основанном на процессах и потоках, код файла redis_washer.py генерирует последовательность посуды:

```
import redis
conn = redis.Redis()
print('Washer is starting')
dishes = ['salad', 'bread', 'entree', 'dessert']
for dish in dishes:
    msg = dish.encode('utf-8')
    conn.rpush('dishes', msg)
    print('Washed', num)
conn.rpush('dishes', 'quit')
print('Washer is done')
```

Цикл генерирует четыре сообщения, содержащие названия тарелок, за которыми следуют финальные сообщения, которые содержат слово quit. Каждое сообщение добавляется в список тарелок на сервере Redis по принципу, сходному с принципами Python.

Как только первая тарелка готова, в работу вступает код файла redis_dryer.py:

```
import redis
conn = redis.Redis()
print('Dryer is starting')
```

```
while True:
    msg = conn.blpop('dishes')
    if not msg:
        break
    val = msg[1].decode('utf-8')
    if val == 'quit':
        break
    print('Dried', val)
print('Dishes are dried')
```

Этот код ожидает прихода сообщений, чьим первым токеном является слово dishes, и выводит сообщение каждой высушенной тарелки. Он подчиняется сообщению quit, завершая цикл.

Запустите сначала сушильщика, а затем мойщика. С помощью символа & в конце команды мы запускаем первую программу в фоновом режиме, она продолжает работать, но больше не принимает команды с клавиатуры. Это работает для операционных систем Linux, OS X и Windows, однако вы можете получить разные результаты в следующей строке. В нашем случае (OS X) этим результатом является некоторая информация о фоновом процессе сушильщика. Далее мы запускаем процесс мойщика как обычно (на переднем плане). Вы увидите смешанную выходную информацию двух процессов:

```
$ python redis_dryer.py &
[2] 81691
Dryer is starting
$ python redis_washer.py
Washer is starting
Washed salad
Dried salad
Washed bread
Dried bread
Washed entree
Dried entree
Washed dessert
Washer is done
Dried dessert
Dishes are dried
[2]+  Done                    python redis_dryer.py
```

Как только идентификаторы посуды начинают приходить от мойщика, наш трудолюбивый процесс сушильщика начинает их обрабатывать. Каждый идентификатор посуды, за исключением финального контрольного значения, является числом из строки quit. Когда процесс сушильщика считает этот идентификатор quit, он завершает работу, после чего в терминал выводится еще немного информации о фоновом процессе (также зависит от системы). Вы можете использовать контрольное значение (некорректное значение), чтобы указать на что-то особенное в потоке данных — в на-

шем случае мы говорим, что закончили работу. В противном случае нам придется добавлять больше программной логики наподобие следующей.

- Заранее оговорить некоторое максимальное количество посуды, что также будет похоже на контрольное значение.
- Выполнять некоторую специфическую коммуникацию вне потока данных между процессами.
- Завершать работу по прошествии какого-то времени, если данных не поступало.

Внесем еще несколько изменений.

- Создадим несколько процессов-сушильщиков.
- Заставим их завершаться по прошествии некоторого времени, если данных не приходило.

Новый файл redis_dryer2.py:

```
def dryer():
    import redis
    import os
    import time
    conn = redis.Redis()
    pid = os.getpid()
    timeout = 20
    print('Dryer process %s is starting' % pid)
    while True:
        msg = conn.blpop('dishes', timeout)
        if not msg:
            break
        val = msg[1].decode('utf-8')
        if val == 'quit':
            break
        print('%s: dried %s' % (pid, val))
        time.sleep(0.1)
    print('Dryer process %s is done' % pid)
import multiprocessing
DRYERS=3
for num in range(DRYERS):
    p = multiprocessing.Process(target=dryer)
    p.start()
```

Запустим процессы сушильщиков в фоновом режиме и процесс мойщика на переднем плане:

```
$ python redis_dryer2.py &
Dryer process 44447 is starting
Dryer process 44448 is starting
Dryer process 44446 is starting
$ python redis_washer.py
```

```
Washer is starting
Washed salad
44447: dried salad
Washed bread
44448: dried bread
Washed entree
44446: dried entree
Washed dessert
Washer is done
44447: dried dessert
```

Один процесс сушильщика считывает идентификатор quit и завершает работу:

```
Dryer process 44448 is done
```

После 20 секунд другие процессы-сушильщики получают значение None от вызова blpop, что указывает на то, что они завершились по таймеру. Они выводят свои последние сообщения и завершаются:

```
Dryer process 44447 is done
Dryer process 44446 is done
```

После того как последний подпроцесс-сушильщик завершается, заканчивается и основная программа-сушильщик:

```
[1]+  Done                    python redis_dryer2.py
```

Помимо очередей

С увеличением числа работающих элементов повышается вероятность того, что что-то может помешать работе нашего конвейера. Если нам нужно помыть посуду после банкета, хватит ли нам работников? Что, если сушильщики напьются до чертиков? Что, если забьется раковина? Ох уж эти проблемы!

Как же справиться с такими проблемами? К счастью, вам доступны некоторые приемы.

- Запустить и забыть. Просто передавайте обработанные объекты дальше и не заботьтесь о последствиях, даже если рядом никого нет. Этот подход похож на сбрасывание посуды на пол.
- Запрос — ответ. Мойщик получает подтверждение от сушильщика, а сушильщик — от того, кто откладывает посуду в сторону. Все это выполняется для каждой тарелки.
- Регулирование нагрузки. Этот прием указывает самому быстрому работнику притормозить, если один из работников, стоящих после него, не поспевает за ним.

В реальных системах вам нужно внимательно следить за тем, чтобы все работники успевали за предложением, в противном случае вы услышите звук бьющейся

посуды. Вы можете добавить новые задачи в список ожидания, а какой-нибудь процесс будет доставать из этого списка последнее сообщение и помещать в список обработки. Когда сообщение будет обработано, оно будет удалено из списка обработки и добавлено в список завершенных задач. Это позволит вам узнать, какие задачи были провалены или занимают слишком много времени. Вы можете сделать это самостоятельно с помощью Redis или использовать систему, которую кто-то написал и протестировал до вас. Некоторые основанные на Python пакеты для работы с очередями (часть из них используют Redis) позволяют удобно управлять процессом.

- Celery. На этот пакет стоит обратить внимание. Он может выполнять распределенные задачи как синхронно, так и асинхронно, используя рассмотренные нами методы: multiprocessing, gevent и др.
- thoonk. Этот пакет создан на базе Redis, он позволяет использовать очереди задач и механизм публикации-подписки (этот механизм будет рассмотрен в следующем разделе).
- rq. Это библиотека Python для очередей задач, она также основана на Redis.
- Queues. Этот сайт предлагает поучаствовать в дискуссии о программном обеспечении для создания очередей, как написанном на Python, так и ином.

Сети

Когда мы говорили о конкуренции, то рассматривали только вопросы, связанные с временем, — решения для одной машины (процессы, потоки, зеленые потоки). Мы также кратко коснулись некоторых решений, которые могут охватывать всю сеть (Redis, ZeroMQ). Теперь же рассмотрим работу с сетями и распределение вычислений в пространстве.

Шаблоны

Сетевые приложения можно создать на основе некоторых простых шаблонов.

Самым распространенным шаблоном является «запрос — ответ», также известный как клиент-сервер. Этот шаблон работает синхронно: клиент ожидает ответа сервера. Вы видели множество примеров использования такого шаблона в этой книге. Ваш браузер — это также клиент, делающий HTTP-запрос к веб-серверу, который возвращает ответ.

Еще одним распространенным шаблоном является «разветвление на выходе»: вы отправляете данные любому доступному работнику из пула процессов. Примером является веб-сервер, расположенный за балансировщиком нагрузки.

Противоположностью этого шаблона является шаблон «разветвление на входе»: вы принимаете данные из одного или более источников. Примером является

регистратор, который принимает текстовые сообщения от нескольких процессов и записывает их в единый файл журнала.

Еще один шаблон похож на радио- или телепередачи, он называется «публикация — подписка», или pub-sub. В рамках этого шаблона публикатор рассылает данные. В простой системе их получат все подписчики. Однако зачастую подписчики могут указать, что они заинтересованы только в определенных типах данных (такие типы часто называются темами), и публикатор будет отправлять только такую информацию. Поэтому, в отличие от шаблона «разветвление на входе», заданный фрагмент данных может получить более чем один подписчик. Если на тему не подписался никто, данные будут проигнорированы.

Модель публикации-подписки

Модель публикации-подписки не является очередью — это широковещательная система. Один (или более) процесс публикует сообщения. Каждый процесс-подписчик указывает, сообщения какого типа он хочет получать. Копия каждого сообщения отправляется каждому подписчику, указавшему этот тип. Поэтому некоторое сообщение может быть обработано однажды, более чем однажды или ни разу. Каждый публикатор просто выполняет рассылку и не знает, кто — если таковые есть — его слушает.

Redis

Вы можете создать быструю систему pub-sub с использованием Redis. Публикатор создает сообщения, имеющие тему и значение, а подписчики указывают, какие темы они хотят получать.

Так выглядит публикатор `redis_pub.py`:

```
import redis
import random
conn = redis.Redis()
cats = ['siamese', 'persian', 'maine coon', 'norwegian forest']
hats = ['stovepipe', 'bowler', 'tam-o-shanter', 'fedora']
for msg in range(10):
    cat = random.choice(cats)
    hat = random.choice(hats)
    print('Publish: %s wears a %s' % (cat, hat))
    conn.publish(cat, hat)
```

Каждая тема представляет собой породу кота, а сопутствующее значение — вид шляпы.

Так выглядит подписчик `redis_sub.py`:

```
import redis
conn = redis.Redis()
```

```
topics = ['maine coon', 'persian']
sub = conn.pubsub()
sub.subscribe(topics)
for msg in sub.listen():
    if msg['type'] == 'message':
        cat = msg['channel']
        hat = msg['data']
        print('Subscribe: %s wears a %s' % (cat, hat))
```

Подписчик показывает, что он хочет принимать сообщения о котах породы `'maine coon'` и `'persian'` и никаких других. Метод `listen()` возвращает словарь. Если он имеет тип `'message'`, это значит, что он был отправлен публикатору и совпадает с нашими критериями. Ключ `'channel'` — это тема (порода кота), а ключ `'data'` содержит сообщение (шляпа).

Если вы сначала запустите публикатор, которого никто не будет слушать, он будет похож на мима, который упал в лесу (издаст ли он звук?), поэтому сначала запустим подписчиков:

```
$ python redis_sub.py
```

Затем запустим публикатор. Он отправит десять сообщений, а затем завершит работу:

```
$ python redis_pub.py
Publish: maine coon wears a stovepipe
Publish: norwegian forest wears a stovepipe
Publish: norwegian forest wears a tam-o-shanter
Publish: maine coon wears a bowler
Publish: siamese wears a stovepipe
Publish: norwegian forest wears a tam-o-shanter
Publish: maine coon wears a bowler
Publish: persian wears a bowler
Publish: norwegian forest wears a bowler
Publish: maine coon wears a stovepipe
```

Подписчика интересуют только две породы котов:

```
$ python redis_sub.py
Subscribe: maine coon wears a stovepipe
Subscribe: maine coon wears a bowler
Subscribe: maine coon wears a bowler
Subscribe: persian wears a bowler
Subscribe: maine coon wears a stovepipe
```

Мы не указали подписчику завершить работу, поэтому он все еще ждет сообщений. Если вы перезапустите публикатор, подписчик получит еще несколько сообщений и выведет их на экран.

Вы можете создать любое количество подписчиков и публикаторов. Если для какого-то сообщения подписчика не найдется, оно пропадет с сервера Redis. Но если подписчики есть, сообщение останется на сервере, пока все подписчики не получат его.

ZeroMQ

Помните сокеты PUB и SUB от ZeroMQ, которые мы видели несколько страниц назад? Они предназначены именно для этого. У ZeroMQ нет центрального сервера, поэтому каждый публикатор пишет всем подписчикам. Перепишем наш пример для ZeroMQ. Публикатор zmq_pub.py выглядит так:

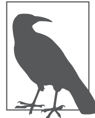
```
import zmq
import random
import time
host = '*'
port = 6789
ctx = zmq.Context()
pub = ctx.socket(zmq.PUB)
pub.bind('tcp://%s:%s' % (host, port))
cats = ['siamese', 'persian', 'maine coon', 'norwegian forest']
hats = ['stovepipe', 'bowler', 'tam-o-shanter', 'fedora']
time.sleep(1)
for msg in range(10):
    cat = random.choice(cats)
    cat_bytes = cat.encode('utf-8')
    hat = random.choice(hats)
    hat_bytes = hat.encode('utf-8')
    print('Publish: %s wears a %s' % (cat, hat))
    pub.send_multipart([cat_bytes, hat_bytes])
```

Обратите внимание на то, как в этом коде используется кодировка UTF-8 для темы и строки значения.

Файл подписчика называется zmq_sub.py:

```
import zmq
host = '127.0.0.1'
port = 6789
ctx = zmq.Context()
sub = ctx.socket(zmq.SUB)
sub.connect('tcp://%s:%s' % (host, port))
topics = ['maine coon', 'persian']
for topic in topics:
    sub.setsockopt(zmq.SUBSCRIBE, topic.encode('utf-8'))
while True:
    cat_bytes, hat_bytes = sub.recv_multipart()
    cat = cat_bytes.decode('utf-8')
    hat = hat_bytes.decode('utf-8')
    print('Subscribe: %s wears a %s' % (cat, hat))
```

В этом коде мы подписываемся на два разных байтовых значения: две строки из `topics`, закодированные с помощью UTF-8.



Это может показаться немного запутанным, но если вы хотите подписываться на все темы, то нужно подписаться на пустую строку байтов `' '`. Если вы этого не сделаете, то не получите ничего.

Обратите внимание на то, что в публикаторе мы вызываем метод `send_multipart()`, а в подписчике — `recv_multipart()`. Это позволяет нам отправлять многокомпонентные сообщения и использовать первую часть как тему. Мы также можем отправить тему и сообщение как простую строку или строку байтов, но подход, где коты и шляпы разделены, кажется более чистым.

Запустите подписчик:

```
$ python zmq_sub.py
```

Запустите публикатор. Он немедленно отправит десять сообщений, а затем завершит работу:

```
$ python zmq_pub.py
Publish: norwegian forest wears a stovepipe
Publish: siamese wears a bowler
Publish: persian wears a stovepipe
Publish: norwegian forest wears a fedora
Publish: maine coon wears a tam-o-shanter
Publish: maine coon wears a stovepipe
Publish: persian wears a stovepipe
Publish: norwegian forest wears a fedora
Publish: norwegian forest wears a bowler
Publish: maine coon wears a bowler
```

Подписчик выведет на экран все, что он запросил и получил:

```
Subscribe: persian wears a stovepipe
Subscribe: maine coon wears a tam-o-shanter
Subscribe: maine coon wears a stovepipe
Subscribe: persian wears a stovepipe
Subscribe: maine coon wears a bowler
```

Другие инструменты для организации подхода публикации-подписки

Вам могут пригодиться следующие ссылки.

- RabbitMQ. Это широко известный брокер сообщений, Python API для него называется `pika`. Обратитесь к документации для `pika` (http://pika.readthedocs.org/) и обучению работе с механизмом публикации — подписки (http://bit.ly/pub-sub-tut).
- pypi.python.org. В правом верхнем углу окна поиска введите `pubsub`, чтобы найти пакеты для Python вроде `pypubsub` (http://pubsub.sourceforge.net/).

- Pubsubhubbub. Этот протокол со сладкозвучным именем позволяет подписчикам зарегистрировать функции обратного вызова для публикаторов (https://code.google.com/p/pubsubhubbub/).

TCP/IP

Мы бродили по дому работы с сетями, принимая за данность, что все, что находится в подвале, работает корректно. Теперь заглянем в подвал и посмотрим на провода и трубы, благодаря которым наверху все работает.

Интернет основан на правилах, по которым создаются соединения, идет обмен данными, закрываются соединения, обрабатывается истечение срока действия и т. д. Эти правила называются протоколами, и они упорядочены в слои. Цель существования слоев заключается в том, чтобы дать путь инновациям и альтернативным способам выполнения задач. Вы можете делать все что угодно в рамках одного слоя до тех пор, пока следуете соглашениям, связанным с работой с более низкими и более высокими слоями.

Самый нижний слой управляет такими аспектами, как электрические сигналы, каждый более высокий слой базируется на нижних. Примерно в середине находится свой IP (Internet Protocol, интернет-протокол), на котором определяется, как адресуются локации сети и как перемещаются пакеты (фрагменты) данных. На слое, который располагается выше его, два протокола описывают, как перемещать байты между локациями.

- UDP (User Datagram Protocol, протокол датаграмм пользователя). Используется для обмена небольшим объемом данных. Датаграмма — это небольшое сообщение, которое отправляется целиком.
- TCP (Transmission Control Protocol, протокол управления передачей). Этот протокол используется для более длинных соединений. С его помощью отправляются потоки байтов, он гарантирует, что они придут по порядку и без дупликаций.

Для сообщений, отправляемых по протоколу UDP, подтверждение не требуется, поэтому вы никогда не можете быть уверены в том, что они придут в точку назначения. Если бы вы хотели рассказать шутку по протоколу UDP, это выглядело бы так:

```
Вот шутка про UDP. Дошло?
```

TCP устанавливает секретное рукопожатие между отправляющей и принимающей стороной, чтобы гарантировать хорошее соединение. Шутка, отправленная по протоколу TCP, начнется примерно так:

```
Ты хочешь услышать шутку про TCP?
Да, я хочу услышать шутку про TCP.
О'кей, я расскажу тебе шутку про TCP.
О'кей, я выслушаю шутку про TCP.
```

```
O'кей, теперь я отправлю тебе шутку про TCP.
O'кей, теперь я приму шутку про TCP.
… (и т. д.)
```

Ваша локальная машина всегда будет иметь IP-адрес `127.0.0.1` и имя `localhost`. Вы можете встретить название для этого процесса — интерфейс обратной петли. Если вы подключены к Интернету, у вашей машины также появится публичный IP. Если же вы используете домашний компьютер, он будет скрыт за оборудованием вроде кабельного модема или роутера. Вы можете запускать интернет-протоколы даже между процессами на одной машине.

Бóльшая часть Интернета, с которой мы будем взаимодействовать, — Всемирная паутина, серверы баз данных и т. д. — основана на протоколе TCP, работающем поверх протокола IP, для краткости — TCP/IP. Сначала рассмотрим самые простые интернет-сервисы. После этого мы рассмотрим общие шаблоны для работы с сетями.

Сокеты

Я приберегал данную тему до этого момента, поскольку вам не нужно знать все низкоуровневые детали для того, чтобы использовать более высокие уровни сети Интернет. Но если вы хотите узнать, как все работает, этот раздел для вас.

На самом низком уровне сетевого программирования используется сокет, позаимствованный из языка программирования C и операционной системы Unix. Кодирование на уровне сокетов может быть утомительным. Вам будет гораздо интереснее работать с чем-то вроде ZeroMQ, однако вам не помешает узнать, что лежит под этим уровнем. Например, сообщения о сокетах часто появляются при возникновении ошибок в работе с сетями.

Напишем небольшой пример, где клиент и сервер будут обмениваться данными. Клиент отправляет серверу строку, размещенную в датаграмме UDP, а сервер возвращает пакет данных, содержащий строку. Серверу нужно слушать определенный адрес и порт — они похожи на почтовое отделение и почтовый ящик. Клиенту нужно знать эти два значения, чтобы доставить сообщение и получить ответ.

В следующем коде клиента и сервера `address` — это кортеж вида `(адрес, порт)`. `address` является строкой, которая может быть именем или IP-адресом. Когда ваши программы просто беседуют друг с другом на одной машине, вы можете использовать имя `'localhost'` или эквивалентный адрес `'127.0.0.1'`.

Для начала отправим небольшой фрагмент данных из одного процесса в другой и вернем немного данных отправителю. Первая программа является клиентом, а вторая — сервером. В каждой программе мы выведем на экран время и откроем сокет. Сервер будет слушать подключения к своему сокету, а клиент — писать данные в свой сокет, что передаст сообщение на сервер.

Так выглядит первая программа, `udp_server.py`:

```
from datetime import datetime
import socket
```

```
server_address = ('localhost', 6789)
max_size = 4096
print('Starting the server at', datetime.now())
print('Waiting for a client to call.')
server = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
server.bind(server_address)
data, client = server.recvfrom(max_size)
print('At', datetime.now(), client, 'said', data)
server.sendto(b'Are you talking to me?', client)
server.close()
```

Сервер должен установить сетевое соединение с помощью двух методов, импортированных из пакета socket. Первый метод, socket.socket, создает сокет, а второй, bind, привязывается к нему (слушает любые данные, приходящие на этот IP-адрес и порт). AF_INET означает, что мы создаем интернет-сокет (IP). (Существует и другой тип для Unix domain sockets, но он будет работать только на локальной машине.) SOCK_DGRAM означает, что мы будем отправлять и получать датаграммы — другими словами, станем использовать UDP.

Теперь сервер просто ждет прихода датаграмм (recvfrom). Когда датаграмма появляется, сервер просыпается и получает данные и информацию о клиенте. Переменная client содержит комбинацию адреса и порта, необходимую для получения доступа к клиенту. Сервер завершает работу, отправляя ответ и закрывая соединение.

Взглянем на файл udp_client.py:

```
import socket
from datetime import datetime
server_address = ('localhost', 6789)
max_size = 4096
print('Starting the client at', datetime.now())
client = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
client.sendto(b'Hey!', server_address)
data, server = client.recvfrom(max_size)
print('At', datetime.now(), server, 'said', data)
client.close()
```

Клиент содержит те же методы, что и сервер (за исключением bind()). Клиент отправляет, а затем получает данные, в то время как сервер сначала получает данные.

Сначала запустите сервер в отдельном окне. Он выведет приветственное сообщение и затем спокойно ждет до тех пор, пока клиент не отправит ему данные:

```
$ python udp_server.py
Starting the server at 2014-02-05 21:17:41.945649
Waiting for a client to call.
```

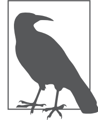
Далее запустим клиент в отдельном окне. Он выведет приветственное сообщение, ответ сервера и завершит работу:

```
$ python udp_client.py
Starting the client at 2014-02-05 21:24:56.509682
At 2014-02-05 21:24:56.518670 ('127.0.0.1', 6789) said b'Are you talking to me?'
```

Наконец, сервер выведет что-то подобное и завершит работу:

```
At 2014-02-05 21:24:56.518473 ('127.0.0.1', 56267) said b'Hey!'
```

Клиенту нужно было знать адрес и номер порта сервера, но он не указал свой номер порта, поэтому тот был автоматически присвоен системой — в этом случае он был равен 56267.



По протоколу UDP данные отправляются небольшими фрагментами. Этот протокол не гарантирует доставки. Если вы отправите несколько сообщений с помощью UDP, они могут прийти в неправильном порядке или вообще не появиться. Этот протокол быстр, легок, не создает соединений, но он ненадежен.

Что приводит нас к протоколу TCP (Transmission Control Protocol, протокол управления передачей). TCP используется для более продолжительных соединений вроде соединений с Интернетом. TCP доставляет данные в том порядке, в котором они были отправлены. Если возникают какие-то проблемы, он пытается отправить их снова. Обменяемся пакетами между клиентом и сервером с помощью TCP.

Файл tcp_client.py действует так же, как и предыдущий клиент, работающий с UDP, отправляя только одну строку на сервер. Однако существуют небольшие различия в вызовах сокетов, показанные здесь:

```
import socket
from datetime import datetime
address = ('localhost', 6789)
max_size = 1000
print('Starting the client at', datetime.now())
client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
client.connect(address)
client.sendall(b'Hey!')
data = client.recv(max_size)
print('At', datetime.now(), 'someone replied', data)
client.close()
```

Мы заменили параметр SOCK_DGRAM на SOCK_STREAM, чтобы получить потоковый протокол, TCP. Мы также добавили вызов connect(), чтобы установить поток. Нам не нужно было делать это для UDP, поскольку каждая датаграмма после отправки предоставлялась сама себе.

Файл tcp_server.py также отличается от своего собрата, работающего с UDP:

```
from datetime import datetime
import socket
address = ('localhost', 6789)
max_size = 1000
print('Starting the server at', datetime.now())
print('Waiting for a client to call.')
server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server.bind(address)
server.listen(5)
client, addr = server.accept()
data = client.recv(max_size)
print('At', datetime.now(), client, 'said', data)
client.sendall(b'Are you talking to me?')
client.close()
server.close()
```

С помощью вызова server.listen(5) сервер конфигурируется так, чтобы поставить в очередь пять клиентских соединений прежде, чем отказывать в следующем. Вызов client.recv(1000) устанавливает максимальную длину входящего сообщения равной 1000 байтам.

Как мы уже делали ранее, запустите сервер, а затем клиент и наблюдайте. Сначала запустим сервер:

```
$ python tcp_server.py
Starting the server at 2014-02-06 22:45:13.306971
Waiting for a client to call.
At 2014-02-06 22:45:16.048865 <socket.socket object, fd=6, family=2, type=1,
    proto=0> said b'Hey!'
```

Теперь запустим клиент. Он отправит сообщение серверу, получит ответ и завершит работу:

```
$ python tcp_client.py
Starting the client at 2014-02-06 22:45:16.038642
At 2014-02-06 22:45:16.049078 someone replied b'Are you talking to me?'
```

Сервер получит сообщение, выведет его на экран, ответит и завершит работу:

```
At 2014-02-06 22:45:16.048865 <socket.socket object, fd=6, family=2, type=1,
    proto=0> said b'Hey!'
```

Обратите внимание на то, что сервер TCP, чтобы ответить, вызвал метод client.sendall(), а в предыдущем примере был вызван метод client.sendto(). TCP поддерживает клиент-серверное соединение с помощью нескольких вызовов сокетов и запоминает IP-адрес клиента.

Это не выглядит ужасно, но если вы попробуете написать что-то более сложное, то увидите, насколько низкоуровневыми являются сокеты. Рассмотрим несколько сложностей, с которыми вам придется столкнуться.

- UDP отправляет сообщения, но их размер ограничен и не гарантируется, что они достигнут места назначения.
- TCP вместо сообщений отправляет потоки байтов. Вы не знаете, сколько байтов отправит или получит система с каждым вызовом.
- Для обмена сообщениями с помощью TCP вам нужна дополнительная информация, чтобы собрать полное сообщение из сегментов: фиксированный размер сообщения (в байтах), или размер всего сообщения, или какой-нибудь разделитель.
- Поскольку сообщения являются байтами, а не текстовыми строками Unicode, вам придется использовать тип `bytes`. Чтобы получить более подробную информацию об этом типе, обратитесь к главе 7.

Если после всего этого вас все еще восхищает программирование сокетов, вам стоит посетить ресурс Python socket programming HOWTO (http://bit.ly/socket-howto), чтобы получить более подробную информацию.

ZeroMQ

Мы уже рассматривали сокеты ZeroMQ, использованные для создания модели публикации-подписки. ZeroMQ является библиотекой. Иногда называемые сокетами на стероидах, сокеты ZeroMQ делают то, чего вы вроде бы ожидаете от обычных сокетов:

- происходит обмен сообщениями целиком;
- выполняются повторные соединения при обрыве;
- выполняется буферизация данных для их сохранения в том случае, когда отправители и получатели не синхронизированы.

Онлайн-руководство (http://zguide.zeromq.org/) написано хорошим языком, в нем представлено лучшее из виденных мной описаний сетевых шаблонов. Питером Хинтдженсом (Pieter Hintjens) создана печатная версия (*ZeroMQ: Messaging for Many Applications*, O'Reilly), внутри которой хороший код, а на обложке — большая рыба (хорошо, что не наоборот). Все примеры в этом печатном руководстве написаны на языке C, но онлайн-версия позволяет вам выбирать один из нескольких языков для каждого примера. Можно даже выбрать примеры для Python (http://bit.ly/zeromq-py). В этой главе я покажу вам базовые приемы работы с ZeroMQ в Python.

ZeroMQ похож на конструктор Lego, и все мы знаем, что даже из небольшого количества деталей можно построить удивительное множество вещей. В этом

случае вы будете создавать сети из сокетов нескольких типов и шаблонов. Основные «детальки Lego», представленные в следующем списке, являются типами сокетов ZeroMQ, которые из-за превратностей судьбы выглядят как шаблоны работы в Сети, рассмотренные нами ранее:

- REQ (синхронный запрос);
- REP (синхронный ответ);
- DEALER (асинхронный запрос);
- ROUTER (асинхронный ответ);
- PUB (публикация);
- SUB (подписка);
- PUSH (разветвление на выходе);
- PULL (разветвление на входе).

Чтобы попробовать поработать с ними самостоятельно, вам нужно установить библиотеку ZeroMQ, введя следующую команду:

```
$ pip install pyzmq
```

Простейший шаблон — это одна пара «запрос — ответ». Он является синхронным: один сокет создает запрос, а затем другой сокет на него отвечает. Сначала рассмотрим код ответа (сервера) zmq_server.py:

```
import zmq
host = '127.0.0.1'
port = 6789
context = zmq.Context()
server = context.socket(zmq.REP)
server.bind("tcp://%s:%s" % (host, port))
while True:
    #  Ожидаем следующего запроса клиента
    request_bytes = server.recv()
    request_str = request_bytes.decode('utf-8')
    print("That voice in my head says: %s" % request_str)
    reply_str = "Stop saying: %s" % request_str
    reply_bytes = bytes(reply_str, 'utf-8')
    server.send(reply_bytes)
```

Мы создаем объект типа Context — это объект ZeroMQ, который обслуживает состояние. Далее создаем сокет ZeroMQ, имеющий тип REP (получено от REPly — ответ). Мы вызываем метод bind(), чтобы заставить его слушать определенный IP-адрес и порт. Обратите внимание на то, что они указаны в строке, 'tcp://localhost:6789', а не в кортеже, как это было в случае с простыми сокетами.

Код в этом примере продолжает получать запросы от отправителя и посылать ответы. Эти сообщения могут быть очень длинными — ZeroMQ обработает детали.

Далее показан код клиента, zmq_client.py. Он имеет тип REQ (получено от REQuest — запрос), в нем вызывается метод connect(), а не bind():

```
import zmq
host = '127.0.0.1'
port = 6789
context = zmq.Context()
client = context.socket(zmq.REQ)
client.connect("tcp://%s:%s" % (host, port))
for num in range(1, 6):
    request_str = "message #%s" % num
    request_bytes = request_str.encode('utf-8')
    client.send(request_bytes)
    reply_bytes = client.recv()
    reply_str = reply_bytes.decode('utf-8')
    print("Sent %s, received %s" % (request_str, reply_str))
```

Пришло время их запустить. Одно интересное отличие от примера с простыми сокетами заключается в том, что вы можете запускать клиент и сервер в любом порядке. Начнем с запуска сервера в одном окне в фоновом режиме:

```
$ python zmq_server.py &
```

Запустите клиент в том же окне:

```
$ python zmq_client.py
```

Вы увидите различающиеся строки от сервера и клиента:

```
That voice in my head says 'message #1'
Sent 'message #1', received 'Stop saying message #1'
That voice in my head says 'message #2'
Sent 'message #2', received 'Stop saying message #2'
That voice in my head says 'message #3'
Sent 'message #3', received 'Stop saying message #3'
That voice in my head says 'message #4'
Sent 'message #4', received 'Stop saying message #4'
That voice in my head says 'message #5'
Sent 'message #5', received 'Stop saying message #5'
```

Наш клиент завершает работу после отправки пятого сообщения, но мы не указали серверу завершить работу, поэтому он будет ожидать новых сообщений. Если вы снова запустите клиент, он выведет те же пять строк, сервер тоже выведет эти пять строк. Если вы не завершите процесс zmq_server.py и попробуете запустить еще один, Python пожалуется на то, что адрес уже используется:

```
$ python zmq_server.py &
[2] 356
Traceback (most recent call last):
  File "zmq_server.py", line 7, in <module>
```

```
    server.bind("tcp://%s:%s" % (host, port))
  File "socket.pyx", line 444, in zmq.backend.cython.socket.Socket.bind
      (zmq/backend/cython/socket.c:4076)
  File "checkrc.pxd", line 21, in zmq.backend.cython.checkrc._check_rc
      (zmq/backend/cython/socket.c:6032)
zmq.error.ZMQError: Address already in use
```

Сообщения нужно отправлять как байтовые строки, поэтому в нашем примере мы закодировали строки в формате UTF-8. Вы можете отправить любое количество сообщений, если будете преобразовывать их в тип bytes. Мы использовали простые текстовые строки как источник сообщений, поэтому методов encode() и decode() будет достаточно, для того чтобы преобразовать их в байтовые строки и обратно. Если ваши сообщения имеют другие типы данных, можете использовать библиотеку вроде MessagePack (http://msgpack.org/).

Даже этот простой шаблон REQ — REP позволяет реализовать некоторые шаблоны коммуникации, поскольку любое количество клиентов REQ может использовать метод connect(), чтобы соединиться с единственным сервером REP. Сервер обрабатывает запросы синхронно по одному за раз, но не сбрасывает другие запросы, ожидающие его внимания. ZeroMQ буферизует сообщения до определенного лимита; именно из-за этого в его имени находится буква Q. Здесь Q расшифровывается как Queue — очередь, M — как Message — сообщение, а Zero (ноль) означает, что ему не нужны посредники.

Несмотря на то что ZeroMQ не предоставляет никаких центральных брокеров (промежуточных участников), вы можете создать их в любой момент. Например, используйте сокеты DEALER и ROUTER, чтобы асинхронно подключиться к нескольким источникам и/или конечным точкам.

Несколько сокетов REQ подключаются к одному сокету ROUTER, который передает каждый запрос сокету DEALER, который, в свою очередь, связывается с подключенным к нему сокетом REP (рис. 11.1). Это похоже на то, как несколько браузеров связываются с прокси-сервером, расположенным перед фермой веб-серверов. Это позволяет вам при необходимости добавить несколько клиентов и серверов.

Сокеты REQ соединяются только с сокетом ROUTER, сокет DEALER соединяется с несколькими сокетами REP, лежащими позади него. ZeroMQ заботится о деталях, гарантируя, что нагрузка, создаваемая запросами, сбалансирована и что ответы будут возвращаться по правильному адресу.

Еще один сетевой шаблон называется «вентилятор», в его рамках используются сокеты PUSH — для того, чтобы перепоручать асинхронные задачи, и сокеты PULL — для того, чтобы собирать результаты.

Последней значимой особенностью ZeroMQ является возможность масштабироваться, просто изменив тип соединения с сокетом при его создании:

- tcp выполняет соединение между процессами на одной или нескольких машинах;
- ipc выполняет соединение между процессами на одной машине;
- inproc выполняет соединение между потоками одного процесса.

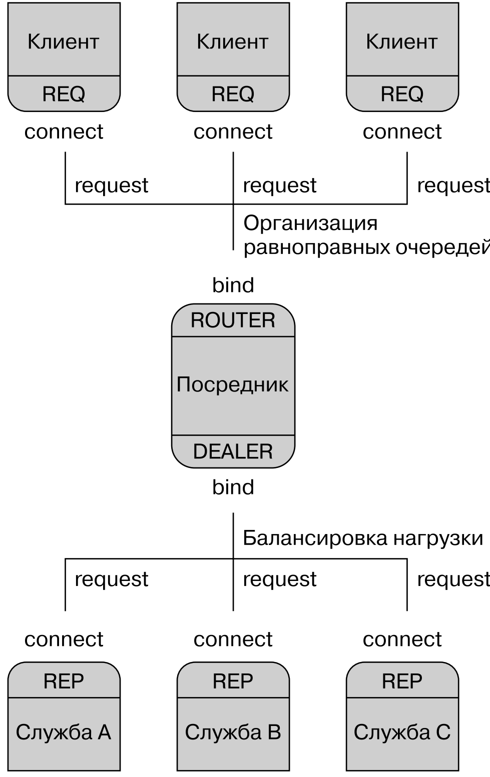


Рис. 11.1. Использование посредника для соединения с несколькими клиентами и серверами

Последнее соединение, `inproc` — это способ передать данные между потоками, избежав блокировок, он является альтернативой примеру работы с потоками, показанному в разделе «Потоки» данной главы.

После использования ZeroMQ вы, возможно, больше не захотите возвращаться к написанию чистого кода для сокетов.

ZeroMQ определенно не единственная библиотека, отвечающая за передачу сообщений, которая поддерживается Python. Передача сообщений — это одна из самых популярных идей в работе с сетями, и Python старается соответствовать другим языкам программирования. Проект Apache, чей веб-сервер вы видели в пункте «Apache» раздела «Веб-сервер» главы 9, также поддерживает проект ActiveMQ (https://activemq.apache.org/), включающий в себя несколько интерфейсов Python, использующих простой текстовый протокол STOMP (http://stomp.github.io/implementations.html). Популярна также библиотека RabbitMQ (http://www.rabbitmq.com/), вы можете прочесть онлайн-руководство для нее (http://bit.ly/rabbitmq-tut).

Scapy

Иногда вам нужно погрузиться в поток данных, путешествующих по сети. Возможно, вы хотите отладить API для веба или отследить какую-то проблему с безопасностью. Библиотека Scapy — это отличный инструмент для того, чтобы исследовать

пакеты, работать с ней гораздо проще, чем писать и отлаживать программы, написанные на языке C. Библиотека является небольшим языком программирования для создания и анализа пакетов.

Я планировал включить в эту книгу примеры исходного кода, но передумал по двум причинам.

- Библиотека Scapy еще не была портирована на Python 3. Раньше это нас не останавливало, мы использовали pip2 и python2, но...
- Инструкция по установке библиотеки Scapy (http://bit.ly/scapy-install), по моему мнению, слишком устрашающая для книги, предназначенной начинающим.

Если хотите, то можете взглянуть на примеры кода на основном сайте с документацией (http://bit.ly/scapy-docs). Они могут вдохновить вас на установку этой библиотеки.

Наконец, не путайте библиотеки Scapy и Scrapy, последняя была рассмотрена в подразделе «Поиск и выборка данных» раздела «Веб-сервисы и автоматизация» главы 9.

Интернет-службы

Python имеет широкий набор инструментов для работы с сетями. В следующих разделах мы рассмотрим способы автоматизации наиболее популярных интернет-служб. В сети доступна полная официальная документация (http://bit.ly/py-internet).

Доменная система имен

Компьютеры имеют числовые IP-адреса вроде 85.2.101.94, но имена мы запоминаем лучше, чем числа. Доменная система имен (Domain Name System, DNS) — это критически важная интернет-служба, которая преобразует IP-адреса в имена и обратно с помощью распределенной базы данных. Когда вы используете браузер и внезапно видите сообщение вроде looking up host, вы, возможно, потеряли соединение с Интернетом, и первым предположением должно стать то, что произошла ошибка DNS.

Некоторые функции DNS можно найти в низкоуровневом модуле socket. Функция gethostbuname() возвращает IP-адрес доменного имени, а ее расширенная версия gethostbyname_ex() возвращает имя, список альтернативных имен и список адресов:

```
>>> import socket
>>> socket.gethostbyname('www.crappytaxidermy.com')
'66.6.44.4'
>>> socket.gethostbyname_ex('www.crappytaxidermy.com')
('crappytaxidermy.com', ['www.crappytaxidermy.com'], ['66.6.44.4'])
```

Метод getaddrinfo() ищет IP-адрес, но также возвращает достаточное количество информации для того, чтобы создать сокет, который с ним соединится:

```
>>> socket.getaddrinfo('www.crappytaxidermy.com', 80)
[(2, 2, 17, '', ('66.6.44.4', 80)), (2, 1, 6, '', ('66.6.44.4', 80))]
```

Предыдущий вызов вернул два кортежа: первый для UDP, а второй — для TCP (6 в строке 2, 1, 6 — это значение для TCP).

Вы можете запросить информацию только для TCP или только для UDP:

```
>>> socket.getaddrinfo('www.crappytaxidermy.com', 80, socket.AF_INET,
socket.SOCK_STREAM)
[(2, 1, 6, '', ('66.6.44.4', 80))]
```

Некоторые номера портов для TCP и UDP (http://bit.ly/tcp-udp-ports) зарезервированы определенными службами IANA и связаны с именами служб. Например, HTTP имеет имя http, ему присвоен номер порта TCP 80.

Эти функции преобразуют имена служб к номерам портов и наоборот:

```
>>> import socket
>>> socket.getservbyname('http')
80
>>> socket.getservbyport(80)
'http'
```

Модули для работы с электронной почтой

Стандартная библиотека Python содержит следующие модули для работы с электронной почтой:

- smtplib — для отправки сообщений по электронной почте с помощью Simple Mail Transfer Protocol (SMTP, простой протокол передачи почты);
- email — для создания и анализа сообщений электронной почты;
- poplib — для чтения электронной почты с помощью Post Office Protocol 3 (POP3, протокол почтового отделения, версия 3);
- imaplib — для чтения электронной почты с помощью Internet Message Access Protocol (IMAP, протокол доступа к электронной почте).

Официальная документация содержит примеры кода (http://bit.ly/py-email) для всех этих библиотек.

Если вы хотите написать собственный SMTP-сервер на Python, попробуйте smtpd (http://bit.ly/py-smtpd).

Написанный на чистом Python SMTP-сервер, который называется Lamson (http://lamsonproject.org/), позволяет хранить сообщения в базе данных, и вы даже сможете блокировать спам.

Другие протоколы

С помощью стандартного модуля ftplib (http://bit.ly/py-ftplib) вы можете перемещать байты с помощью File Transfer Protocol (FTP). Несмотря на свой возраст, протокол FTP все еще хорошо работает.

Вы видели, как эти модули используются повсеместно в разных местах этой книги, взгляните также на документацию, касающуюся поддержки интернет-протоколов в стандартной библиотеке (http://bit.ly/py-internet).

Веб-службы и API

Поставщики информации всегда имеют сайт, но он предназначен для человеческих глаз, а не для машин. Если данные опубликованы только на сайте, любой, кто хочет получить к ним доступ, должен писать краулер (это показано в подразделе «Поиск и выборка данных» раздела «Веб-сервисы и автоматизация» главы 9) и переписывать их после каждого изменения формата. Обычно это утомительно. В противоположность этому, если сайт предлагает API для своих данных, эти данные становятся доступными для клиентских программ. API меняются реже, чем макеты веб-страниц, поэтому изменения в клиентах также меньше распространены. Быстрый чистый конвейер также позволяет проще создавать гибридные приложения — комбинации, которые не предвиделись, но могут быть полезны и даже прибыльны.

Простейшим API является веб-интерфейс, который предоставляет данные в структурированном формате вроде JSON или XML (но не в текстовом формате или формате HTML). API может быть минимальным или полнофункциональным RESTful API (это понятие рассматривается в подразделе «API для Сети и Representational State Transfer» раздела «Веб-сервисы и автоматизация» главы 9), это позволит найти еще один выход для байтов, не знающих устали.

В самом начале этой книги вы видели веб-API — этот интерфейс выбрал самые популярные видеоролики с YouTube. Следующий пример теперь покажется вам более осмысленным, поскольку вы уже прочитали о веб-запросах, JSON, словарях, списках и разбиении:

```
import requests
url = "https://gdata.youtube.com/feeds/api/standardfeeds/top_rated?alt=json"
response = requests.get(url)
data = response.json()
for video in data['feed']['entry'][0:6]:
    print(video['title']['$t'])
```

API особенно полезны для получения данных с популярных сайтов социальных медиа вроде Twitter, Facebook и LinkedIn. Все эти сайты предоставляют бесплатные API, но требуют от вас регистрации и получения ключа (долго генерируемой текстовой строки, ее часто называют токеном), который будет использоваться при соединении. Ключ помогает сайту определить, кто получает доступ к данным. Он также может использоваться для того, чтобы ограничить трафик запросов. Пример с YouTube, который вы только что видели, не требует использования API-ключа для поиска. Однако ключ потребуется, если вы будете делать вызовы, обновляющие данные на YouTube.

У следующих брендов имеются интересные API служб:

- New York Times (http://developer.nytimes.com/);
- YouTube (http://gdata.youtube.com/demo/index.html);

- Twitter (https://dev.twitter.com/docs/twitter-libraries);
- Facebook (https://developers.facebook.com/tools/);
- Weather Underground (http://www.wunderground.com/weather/api/);
- Marvel Comics (http://developer.marvel.com/).

Примеры API для карт вы можете увидеть в приложении Б, примеры других API — в приложении В.

Удаленная обработка

Большинство примеров этой книги показывали, как вызвать код Python на одной машине зачастую в рамках одного процесса. Но Python позволяет вызывать код на других машинах, если они входят в локальную сеть. Сеть компьютеров дает вам доступ к большему количеству процессов и/или потоков.

Удаленные вызовы процедур

Удаленные вызовы процедур (Remote Procedure Call, RPC) выглядят как обычные функции, но выполняются на удаленных машинах по всей сети. Вместо того чтобы вызывать RESTful API и передавать туда аргументы, закодированные в URL или теле запросов, вы можете вызвать функцию RPC на своей собственной машине. При этом в RPC-клиенте произойдет следующее.

1. Он преобразует аргументы вашей функции в байты (иногда это называется маршаллингом, сериализацией или просто кодированием).
2. Он отправляет закодированные байты удаленной машине.

И вот что происходит на удаленной машине.

1. Она получает закодированные байты запроса.
2. После получения байтов RPC-клиент декодирует их в оригинальные структуры данных (или аналогичные, если аппаратное и программное обеспечение двух машин различаются).
3. Затем клиент находит и вызывает локальную функцию с помощью раскодированных данных.
4. Далее он кодирует результат работы функции.
5. Наконец, клиент отправляет закодированные байты вызывающей стороне.

После этого машина, запустившая процесс, декодирует полученные байты в возвращенные значения.

RPC — это популярный прием, и люди реализовали его множеством способов. На стороне сервера вы запускаете серверную программу, создаете механизм для ее связывания с помощью какого-нибудь способа транспортировки байтов и метода кодирования/декодирования, определяете функции службы и включаете питание

знака «RPC готов к работе». Клиенты соединяются с сервером и вызывают одну из его функций с помощью RPC.

Стандартная библиотека содержит только одну реализацию RPC, которая использует в качестве формата обмена данными XML, — xmlrpc. Вы определяете и регистрируете функции на сервере, а клиент вызывает их так, будто они были импортированы. Сначала рассмотрим файл xmlrpc_server.py:

```
from xmlrpc.server import SimpleXMLRPCServer
def double(num):
    return num * 2
server = SimpleXMLRPCServer(("localhost", 6789))
server.register_function(double, "double")
server.serve_forever()
```

Функция, которую мы предоставляем на сервере, называется double(). В качестве аргумента она ожидает число, а возвращает это же число, умноженное на два. Сервер начинает работу на определенных адресе и порте. Нам нужно зарегистрировать функцию, чтобы сделать ее доступной клиентам с помощью RPC. Наконец, можно запустить ее.

Теперь, как вы догадались, рассмотрим файл xmlrpc_client.py:

```
import xmlrpc.client
proxy = xmlrpc.client.ServerProxy("http://localhost:6789/")
num = 7
result = proxy.double(num)
print("Double %s is %s" % (num, result))
```

Клиент соединяется с сервером с помощью функции ServerProxy(). Далее он вызывает функцию proxy.double(). Откуда она появилась? Она была создана динамически с помощью сервера. Механизм RPC волшебным образом прикрепляет имя функции к вызову удаленного сервера.

Попробуйте сами — запустите сервер и клиент:

```
$ python xmlrpc_server.py
```

Далее запустите клиент:

```
$ python xmlrpc_client.py
Double 7 is 14
```

После этого сервер выведет на экран следующее:

```
127.0.0.1 -- [13/Feb/2014 20:16:23] "POST / HTTP/1.1" 200 -
```

Популярными методами передачи данных являются HTTP и ZeroMQ. Другими распространенными кодировками, помимо XML, являются JSON, Protocol Buffers и MessagePack. Существует множество пакетов для работы с RPC, использующих JSON, но многие из них либо не поддерживают Python 3, либо кажутся несколько запутанными. Взглянем на кое-что другое — реализацию Python

RPC в рамках MessagePack (http://bit.ly/msgpack-rpc). Установить ее можно следующим образом:

```
$ pip install msgpack-rpc-python
```

Этот вызов также установит tornado, написанный на Python веб-сервер, основанный на событиях, который эта библиотека использует как транспорт. Как обычно, рассмотрим сервер первым (msgpack_server.py):

```
from msgpackrpc import Server, Address
class Services():
    def double(self, num):
        return num * 2
server = Server(Services())
server.listen(Address("localhost", 6789))
server.start()
```

Методы класса Services доступны благодаря RPC. Рассмотрим клиент msgpack_client.py:

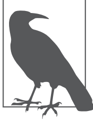
```
from msgpackrpc import Client, Address
client = Client(Address("localhost", 6789))
num = 8
result =  client.call('double', num)
print("Double %s is %s" % (num, result))
```

Для того чтобы запустить этот код, следуйте обычным инструкциям — запустите сервер, запустите клиент, посмотрите на результат:

```
$ python msgpack_server.py
$ python msgpack_client.py
Double 8 is 16
```

fabric

Пакет fabric позволяет вам запускать удаленные или локальные команды, загружать или закачивать файлы и работать от лица привилегированного пользователя с помощью команды sudo. Пакет использует Secure Shell (SSH, зашифрованный текстовый протокол, заменивший telnet) для того, чтобы запускать программы на удаленных машинах. Вы пишете функции (на Python) в так называемом файле fabric и указываете, как их нужно запустить — локально или удаленно. Когда вы запустите эти файлы с помощью программы fabric (которая называется fab и не является отсылкой к Beatles или моющему веществу), вы указываете, какие удаленные машины нужно использовать и какие функции вызывать. Это проще, чем примеры RPC, которые мы рассмотрели ранее.



На момент написания этой книги автор пакета fabric вносил в свое творение поправки, которые позволят пакету запускаться с помощью Python 3. Если все пройдет успешно, примеры, приведенные далее, будут работать. В противном случае вам придется запускать их с помощью Python 2.

Для начала установим пакет fabric с помощью следующей команды:

```
$ pip2 install fabric
```

Вы можете запустить код локально из файла fabric непосредственно, без использования SSH. Сохраните первый файл под именем fab1.py:

```
def iso():
    from datetime import date
    print(date.today().isoformat())
```

Далее введите следующую команду, чтобы запустить его:

```
$ fab -f fab1.py -H localhost iso
[localhost] Executing task 'iso'
2014-02-22
Done.
```

Опция -f fab1.py указывает использовать файл fabric fab1.py вместо варианта по умолчанию fabfile.py. Опция -H localhost указывает запустить команду на вашем локальном компьютере. Наконец, iso — это имя функции, которую нужно запустить. Она сработает точно так же, как и в рассмотренном нами примере, где использовались RPC. Вы можете найти большее количество опций на сайте с документацией (http://docs.fabfile.org/).

Для того чтобы запускать внешние программы на локальной или удаленной машинах, вам нужно запустить SSH-сервер. В системах семейства Unix этот сервер называется sshd; команда service sshd status скажет вам, запущен ли сервер, а команда service sshd start запустит его при необходимости. В операционных системах Mac откройте пункт меню System Preferences, щелкните на вкладке Sharing, а затем установите флажок Remote Login. Операционная система Windows не имеет встроенной поддержки SSH, вам стоит установить putty (http://bit.ly/putty-ssh).

Сейчас мы снова используем имя функции iso, но в этот раз заставим ее запускать команду с помощью метода local(). Так выглядят команда и результат ее работы:

```
from fabric.api import local
def iso():
    local('date -u')
$ fab -f fab2.py -H localhost iso
 [localhost] Executing task 'iso'
[localhost] local: date -u
Sun Feb 23 05:22:33 UTC 2014
Done.
Disconnecting from localhost... done.
```

Удаленный двойник функции local() — функция run(). Так выглядит файл fab3.py:

```
from fabric.api import run
def iso():
    run('date -u')
```

Применение функции run() указывает fabric использовать SSH для того, чтобы связаться с указанными в командной строке хостами, поскольку была указана опция -H (показано в следующем примере). В противном случае будет использован локальный компьютер, который поведет себя так, будто является удаленной машиной, — это может быть полезно для тестирования. В этом примере мы снова будем использовать локальную машину:

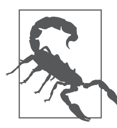
```
$ fab -f fab3.py -H localhost iso
[localhost] Executing task 'iso'
[localhost] run: date -u
[localhost] Login password for 'yourname':
[localhost] out: Sun Feb 23 05:26:05 UTC 2014
[localhost] out:
Done.
Disconnecting from localhost... done.
```

Обратите внимание на то, что мне предложили ввести пароль. Для того чтобы этого избежать, вы можете встроить пароль в файл fabric следующим образом:

```
from fabric.api import run
from fabric.context_managers import env
env.password = "your password goes here"
def iso():
    run('date -u')
```

Теперь запустите файл:

```
$ fab -f fab4.py -H localhost iso
[localhost] Executing task 'iso'
[localhost] run: date -u
[localhost] out: Sun Feb 23 05:31:00 UTC 2014
[localhost] out:
Done.
Disconnecting from localhost... done.
```

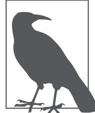


Размещать свой пароль внутри кода ненадежно и небезопасно. Лучший способ указать необходимый пароль — сконфигурировать SSH, задав открытый и закрытый ключи, с помощью ssh-keygen (http://bit.ly/genkeys).

Salt

Salt создавался как способ реализовать удаленное выполнение программ, но позже он вырос в полноценную платформу управления системами. Основанный на ZeroMQ вместо SSH, он может работать с тысячами серверов.

Salt еще не был портирован на Python 3. В этом случае я не стану показывать вам примеры, написанные на Python 2. Если вам интересна эта область, прочтите документацию и ждите объявлений о том, когда закончится портирование.



Альтернативными продуктами являются puppet (http://puppetlabs.com/) и chef (http://www.getchef.com/chef/), тесно связанные с Ruby. Пакет ansible (http://www.ansible.com/home), который, как и salt, написан с помощью Python, также можно поставить с ними в один ряд. Вы можете загрузить и использовать его бесплатно, но поддержка и некоторые пакеты с надстройками требуют коммерческой лицензии. По умолчанию он использует SSH и не требует установки особого программного обеспечения на тех компьютерах, которыми будет управлять.

Пакеты salt и ansible функционально являются супермножествами пакета fabric, поскольку они обрабатывают исходную конфигурацию, развертывание и удаленное выполнение.

Большие данные и MapReduce

По мере роста Google и других интернет-компаний обнаружилось, что традиционные вычислительные решения не масштабировались. Программное обеспечение, которое работало на отдельных или даже на нескольких машинах, не могло справиться с тысячами.

Из-за объемов дискового пространства для баз данных и файлов для поиска требовалось множество механических движений дисковой головки. (Подумайте о виниловой пластинке и времени, которое требуется для того, чтобы переместить иголку с одной дорожки на другую вручную. А также подумайте о скрипящем звуке, который она издаст, если вы надавите слишком сильно, не говоря уже о звуках, которые издаст хозяин пластинки.) Но передавать потоком последовательные фрагменты диска вы можете быстрее.

Разработчики обнаружили, что гораздо быстрее было распространять и анализировать данные на нескольких машинах, объединенных в сеть, чем на отдельных. Они могли использовать алгоритмы, которые звучали просто, но на деле в целом лучше работали с объемными распределенными данными. Один из таких алгоритмов называется MapReduce, он может распределить вычисления между несколькими компьютерами и затем собрать результат. Это похоже на работу с очередями.

После того как компания Google опубликовала полученные результаты, компания Yahoo! вслед за ней создала пакет с открытым исходным кодом, написанный на Java, который называется Hadoop (назван в честь игрушечного плюшевого слона, принадлежавшего сыну главного разработчика).

Здесь вступают в действие слова «большие данные». Зачастую это просто означает, что «данных слишком много, чтобы они поместились на мою машину»: данные, объем которых превышает дисковое пространство, память, время работы процессора или все перечисленное. Для некоторых организаций решением вопроса больших данных является Hadoop. Hadoop копирует данные среди машин, пропускает их через программы масштабирования и сжатия и сохраняет на диск результаты после каждого шага.

Этот процесс может быть медленным. Более быстрым методом является отправка потоком с помощью Hadoop, которая работает как каналы Unix, посылая данные между программами и не требуя записи на диск после выполнения каждо-

го шага. Вы можете писать программы, использующие отправку потоком с помощью Hadoop, на любом языке, включая Python.

Множество модулей Python были написаны для Hadoop, некоторые из них рассматриваются в статье блога A Guide to Python Frameworks for Hadoop (http://bit.ly/py-hadoop). Компания Spotify, известная передачей потоковой музыки, открыла исходный код своего компонента для отправки потоком с помощью Hadoop, Luigi, написанного на Python. Порт для Python 3 все еще не готов.

Конкурент по имени Spark (http://bit.ly/about-spark) был разработан для того, чтобы превысить скорость работы Hadoop в 10–100 раз. Он может читать и обрабатывать любой источник данных и формат Hadoop. Spark включает в себя API для Python и других языков. Вы можете найти документацию по установке онлайн (http://bit.ly/dl-spark).

Еще одной альтернативой Hadoop является Disco (http://discoproject.org/), который использует Python для обработки MapReduce и язык программирования Erlang для коммуникации. К сожалению, вы не можете установить его с помощью pip — загляните в документацию (http://bit.ly/get-disco).

Обратитесь к приложению В, чтобы увидеть связанные с нашей темой примеры параллельного программирования, в которых объемные структурированные вычисления распространены между несколькими машинами.

Работаем в облаках

Не так давно вам приходилось покупать собственные серверы, размещать их на стойках в дата-центрах и устанавливать на них множество программ: операционные системы, драйверы устройств, файловые системы, базы данных, веб-серверы, серверы электронной почты, балансировщики нагрузки, мониторы и т. д. Эффект новизны пропал, когда вы начали пытаться поддерживать работоспособность нескольких систем. Кроме того, вам приходилось постоянно волноваться о безопасности.

Многие хостинговые службы предлагали позаботиться о ваших серверах за некоторую плату, но вам все равно приходилось брать в аренду физические устройства и постоянно платить за пиковую нагрузку.

С увеличением количества компьютеров ошибки становились все более распространенными. Вам нужно было масштабировать службы горизонтально и хранить избыточные данные. Вы не можете предполагать, что сеть будет работать как одна машина. Согласно Питеру Дойчу (Peter Deutsch), восемь ошибок восприятия распределенных вычислений заключаются в следующем.

- Сеть надежна.
- Латентность равна нулю.
- Полоса пропускания бесконечна.
- Сеть безопасна.

- Топология не меняется.
- Существует всего один администратор.
- Стоимость транспортировки равна нулю.
- Сеть гомогенна.

Вы можете попробовать создать сложную распределенную систему, но для этого потребуется много работы и другой набор инструментов. Можно сравнить серверы с домашними животными — вы даете им имена, знаете их характер и лечите их по мере необходимости. Но по мере роста их количества они становятся больше похожими на скот: выглядят одинаково, имеют номера, и их просто заменяют, если возникает какая-то проблема.

Вместо того чтобы создавать систему, вы можете арендовать сервер в облаке. В рамках этой модели обслуживание серверов является заботой кого-то другого, а вы можете сконцентрироваться на своей службе, блоге или чем-то другом, что хотели бы показать миру. Используя онлайн-панель инструментов и API, вы можете выбрать серверы с любой необходимой вам конфигурацией быстро и легко — они эластичны. Вы можете отслеживать их статус и получать предупреждения в том случае, если какой-то показатель превысил лимит. Облака в данный момент являются довольно популярной темой, и корпоративные расходы на облачные компоненты все больше растут.

Рассмотрим, как Python взаимодействует с некоторыми популярными облаками.

Google

Google часто использует Python для внутренних нужд и нанимает именитых разработчиков Python (у них какое-то время работал сам Гвидо ван Россум).

Перейдите на сайт App Engine (https://developers.google.com/appengine/) и затем в меню Choose a Language (Выберите язык) установите флажок Python. Вы можете вводить код Python в Cloud Playground и видеть результаты внизу. Сразу после этого раздела вы увидите ссылки, с помощью которых можете загрузить на свой компьютер Python SDK. Это позволит вам разрабатывать для облачных API от Google API на собственном компьютере. Далее рассмотрим детали развертывания вашего приложения на AppEngine.

На главной странице Google (https://cloud.google.com/), касающейся темы облаков, вы можете найти детали о его службах, включая следующие:

- App Engine — высокоуровневая платформа, включающая такие инструменты Python, как Flask и django;
- Compute Engine — создает кластеры виртуальных машин для объемных задач, связанных с распределенными вычислениями;
- Cloud Storage — хранилище объектов (объектами являются файлы, здесь нет иерархий каталогов);
- Cloud Datastore — крупная база данных NoSQL;

- Cloud SQL — крупная база данных SQL;
- Cloud Endpoints — обеспечивает доступ к приложениям с помощью Restful;
- BigQuery — большие данные в стиле Hadoop.

Службы Google конкурируют со службами компаний Amazon и OpenStack.

Amazon

По мере роста компании Amazon от сотен до тысяч и миллионов серверов разработчики столкнулись со всеми проблемами распределенных систем. Однажды в 2002 году (или около того) CEO компании Джефф Безос (Jeff Bezos) объявил работникам Amazon, что с этого момента все данные и функционал должны быть доступны только через интерфейсы сетевых служб — не через файлы, базы данных или локальные вызовы функций. Им пришлось разрабатывать эти интерфейсы так, как если бы их код стал общедоступным. Письмо заканчивалось мотивирующей фразой: «Тот, кто этого не сделает, будет уволен».

Неудивительно, что разработчики взялись за дело и с течением времени создали очень крупную архитектуру, ориентированную на службы. Они позаимствовали или придумали сами множество решений, включая Amazon Web Services (AWS) (http://aws.amazon.com/), которое сейчас доминирует на рынке. Теперь оно состоит из множества служб, но самыми важными являются следующие:

- Elastic Beanstalk — высокоуровневая платформа для приложений;
- EC2 (Elastic Compute) — распределенные вычисления;
- S3 (Simple Storage Service) — хранилище объектов;
- RDS — реляционные базы данных (MySQL, PostgreSQL, Oracle, MSSQL);
- DynamoDB — база данных NoSQL;
- Redshift — хранилище данных;
- EMR;
- Hadoop.

Чтобы подробнее узнать об этих и других службах AWS, загрузите Amazon Python SDK (http://bit.ly/aws-py-sdk) и прочтите раздел Помощь.

Официальная библиотека AWS для Python, boto (http://docs.pythonboto.org/), также все еще не полностью портирована на Python 3. Вам понадобится использовать Python 2 или искать альтернативу, что вы можете сделать, введя в строку поиска Python Package Index (http://pypi.python.org/) `aws` или `amazon`.

OpenStack

Вторым самым популярным облачным сервисом был Rackspace. В 2010 году компания создала необычное партнерство с NASA, чтобы объединить свои части облачной инфраструктуры в OpenStack (http://www.openstack.org/). Это бесплатное решение с открытым исходным кодом, которое может использоваться для создания открытых, закрытых и гибридных облаков. Новая версия выходит каждые шесть месяцев,

самая последняя из них на момент написания книги содержала более 1,25 млн строк кода Python, внесенного многими участниками. OpenStack используется на производстве постоянно увеличивающимся числом организаций, куда входят CERN и PayPal.

Основные API OpenStack являются RESTful, модули Python предоставляют программные интерфейсы, а программы, запускающиеся из командной строки, отвечают за автоматизацию работы с оболочкой. Рассмотрим стандартные службы текущей версии:

- Keystone — служба идентификации, предоставляющая аутентификацию (например, логин/пароль), авторизацию (инструменты) и обнаружение служб;
- Nova — служба вычислений, распределенная работа с серверами сети;
- Swift — хранилище объектов вроде S3 от Amazon. Оно используется службой Cloud Files от Rackspace;
- Glance — служба хранения изображений промежуточного уровня;
- Cinder — служба хранения блоков низкого уровня;
- Horizon — онлайн-панель управления для всех служб;
- Neutron — служба управления сетью;
- Heat — служба управления (мультиоблачная);
- Ceilometer — служба телеметрии (метрики, мониторинг и измерения).

Время от времени появляются и другие сервисы, затем они проходят через инкубационный процесс и могут стать частью стандартной платформы OpenStack.

OpenStack работает на Linux или внутри виртуальной машины Linux. Установка его основных служб довольно сложна. Самый быстрый способ установить OpenStack на Linux — использовать Devstack (http://devstack.org/) и читать в процессе весь объясняющий текст. В конце вы увидите онлайн-панель инструментов, которая может просматривать другие службы и управлять ими.

Если вы хотите установить некоторые службы или даже их все вручную, используйте менеджер пакетов вашего дистрибутива. Все основные поставщики Linux поддерживают OpenStack и предоставляют пакеты на своих серверах. Посетите сайт OpenStack, чтобы увидеть документы по установке, новости и соответствующую информацию.

Разработка и корпоративная поддержка OpenStack все больше ускоряются. Его начали сравнивать с Linux, когда тот мешал распространению версий Unix.

Упражнения

1. Используйте объект класса `socket`, чтобы реализовать службу, сообщающую текущее время. Когда клиент отправляет на сервер строку `'time'`, верните текущие дату и время как строку ISO.

2. Используйте сокеты ZeroMQ `REQ` и `REP`, чтобы сделать то же самое.
3. Попробуйте сделать то же самое с помощью `XMLRPC`.
4. Возможно, вы видели эпизод телесериала I Love Lucy, в котором Люси и Этель работают на шоколадной фабрике (это классика). Парочка стала отставать, когда линия конвейера, которая направляла к ним на обработку конфеты, начала работать еще быстрее. Напишите симуляцию, которая отправляет разные типы конфет в список Redis, и клиент `Lucy`, который делает блокирующие выталкивания из списка. Ей нужно 0,5 секунды, чтобы обработать одну конфету. Выведите на экран время и тип каждой конфеты, которую получит `Lucy`, а также количество необработанных конфет.
5. Используйте ZeroMQ, чтобы публиковать стихотворение из упражнения 7 главы 7 по одному слову за раз. Напишите потребителя ZeroMQ, который будет выводить на экран каждое слово, начинающееся с гласной. Напишите другого потребителя, который будет выводить все слова, состоящие из пяти букв. Знаки препинания игнорируйте.

12 Быть питонщиком

> Всегда хотели отправиться во времени назад, чтобы сразиться с более молодой версией себя? Карьера в разработке ПО — это то, что вам нужно!
>
> *Эллиот Лох (Elliot Loh)*

Эта глава посвящена науке и искусству разработки с помощью Python, она содержит рекомендации и правила хорошего тона. Изучите их, и вы тоже сможете стать настоящим питонщиком.

О программировании

Для начала я хочу сказать пару слов о программировании с высоты личного опыта.

Я начинал свою деятельность в области науки и обучился программированию, чтобы анализировать и отображать экспериментальные данные. Мне казалось, что программирование окажется похожим на бухгалтерский учет — будет точным и скучным. Но я удивился, когда понял, что мне это нравится. Одними из интересных для меня аспектов стали логическая — программирование похоже на складывание пазлов — и творческая составляющие. Вам нужно написать программу так, чтобы получить правильные результаты, но вы вольны написать ее тем способом, который вам больше нравится. Такое соотношение использования левого и правого полушарий мозга необычно.

После того как я начал свою карьеру в программировании, я также узнал, что в этой области существует множество ниш для разных задач и разных типов людей. Вы можете погрузиться в область компьютерной графики, операционных систем, бизнес-приложений и даже науки.

Если вы программист, у вас мог быть похожий опыт. Если же нет, можете попробовать начать программировать, чтобы посмотреть, подходит ли это вам или хотя бы помогает ли это решить какие-то задачи. Как я уже писал в этой книге, знание математики здесь не так уж важно. Скорее всего, самое главное — это спо-

собность мыслить логически и склонность к языкам, что может помочь при программировании. Наконец, вам пригодится терпение, особенно если вы отслеживаете баг в своем коде.

Ищем код на Python

Когда вам нужно написать некий код, самым быстрым решением является кража. Конечно же, воровать можно только из тех источников, которые позволяют это делать.

Стандартная библиотека Python (http://docs.python.org/3/library/) широка, глубока и довольно понятна. Погрузитесь в нее и ищите жемчужины.

Как и в случае с залами славы в спорте, модулю требуется время, чтобы попасть в стандартную библиотеку. Новые пакеты появляются довольно часто, и на протяжении этой книги я отмечал те из них, которые либо делают что-то новое, либо делают что-то старое лучше. Python поставляется сразу «с батарейками», но иногда вам нужна батарейка другого вида.

Так где же, помимо стандартной библиотеки, следует искать хороший код Python?

Первое место, на которое вы должны обратить внимание, — это Python Package Index (PyPI) (https://pypi.python.org/pypi). Ранее носивший имя Cheese Shop в честь скетча Monty Python, этот сайт постоянно обновляется — на момент написания этой книги он содержит более 39 000 пакетов. Когда вы используете `pip` (смотрите следующий раздел), он ищет пакет на сайте PyPI. Основная страница PyPI показывает самые свежие пакеты. Вы также можете выполнить прямой поиск. Например, в табл. 12.1 показаны результаты поиска по слову `genealogy`.

Таблица 12.1. Пакеты для работы с генеалогическим древом, которые вы можете найти на сайте PyPI

Пакет	Вес	Описание
Gramps 3.4.2	5	Исследуйте, организуйте и делитесь своей семейной генеалогией
python-fs-stack 0.2	2	Оболочка, написанная на Python, для всех API FamilySearch
human-names 0.1.1	1	Человеческие имена
nameparser 0.2.8	1	Простой модуль Python, предназначенный для разбиения человеческих имен на отдельные компоненты

Лучшие совпадения имеют больший вес, поэтому пакет Gramps, скорее всего, подойдет вам лучше всего. Посетите сайт Python (https://pypi.python.org/pypi/Gramps/3.4.2), чтобы увидеть документацию и ссылки на загрузку пакета.

Еще одним популярным репозиторием является GitHub. Взгляните, какие пакеты Python популярны в данный момент.

Сайт Popular Python recipes (http://bit.ly/popular-recipes) содержит более 4000 коротких программ Python на любую тему.

Установка пакетов

Существует три способа установить пакет Python.

- Использовать pip, если есть такая возможность. С помощью pip вы можете установить большинство пакетов.
- Иногда вы можете использовать менеджер пакетов своей операционной системы.
- С помощью исходного кода.

Если вам нужно установить несколько пакетов из одной области, вы можете обнаружить дистрибутив, который уже содержит их. Например, в приложении В вы можете попробовать поработать с несколькими числовыми и научными программами, которые было бы трудно устанавливать вручную, но все они включены в дистрибутивы вроде Anaconda.

Используем pip

Использование пакетов Python имело несколько ограничений. Более ранний инструмент для установки easy_install был заменен pip, но ни один из них не находился в стандартном пакете Python. Если мы должны устанавливать пакеты с помощью pip, то где же его взять? Начиная с Python 3.4, pip наконец-то включили в стандартный пакет Python, чтобы избежать подобного экзистенциального кризиса. Если вы используете более раннюю версию Python и у вас не установлен pip, можете загрузить его по адресу http://www.pip-installer.org.

Простейший вариант использования pip — установка последней версии некоторого пакета с помощью следующей команды:

```
$ pip install flask
```

Вы увидите детали происходящего, просто чтобы не подумали, что pip ленится: загрузка, запуск setup.py, установка файлов на диск и др.

Вы также можете указать pip установить определенную версию:

```
$ pip install flask==0.9.0
```

Или минимальную версию (это полезно, когда некоторая особенность, без которой вы жить не можете, появляется только в определенной версии):

```
$ pip install 'flask>=0.9.0'
```

В предыдущем примере одинарные кавычки не дают оболочке интерпретировать символ >, чтобы перенаправить поток выходной информации в файл с именем =0.9.0.

Если вы хотите установить более одного пакета, можете воспользоваться файлом требований (http://bit.ly/pip-require). Несмотря на обилие вариантов, простейшим

вариантом использования является список пакетов, в каждой строке по одному, опционально содержащий точные или относительные версии:

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

Например, файл `requirements.txt` может содержать следующее:

```
flask==0.9.0
django
psycopg2
```

Менеджер пакетов

Apple's OS X содержит сторонние менеджеры пакетов homebrew (brew) (http://brew.sh/) и ports (http://www.macports.org/). Они работают примерно так же, как и `pip`, но не ограничены пакетами Python.

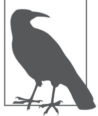
В операционных системах семейства Linux имеется отдельный менеджер пакетов для каждого дистрибутива. Самыми популярными являются `apt-get`, `yum`, `dpkg` и `zypper`.

В операционных системах семейства Windows имеются Windows Installer и файлы пакетов с суффиксом .msi. Если вы устанавливали Python для Windows, то, скорее всего, файл пакета имел формат MSI.

Установка из исходного кода

Иногда случается так, что пакет еще совсем новый или же автор просто не сделал его доступным через `pip`. Для того чтобы создать пакет, вы, как правило, делаете следующее.

1. Загружаете код.
2. Извлекаете файлы с помощью zip, tar или другого подходящего инструмента, если они заархивированы или сжаты.
3. Запускаете команду `python install setup.py` в папке, которая содержит файл `setup.py`.



Как и обычно, вам следует быть осторожными с тем, что вы загружаете и устанавливаете. В программах, написанных на Python, вредоносный код спрятать труднее, поскольку они представляют собой читабельный текст, но иногда это случается.

Интегрированные среды разработки

Для написания программ, представленных в этой книге, я использовал текстовый интерфейс, но это не значит, что вы должны запускать весь код в консоли или текстовом окне. Существует множество бесплатных и коммерческих интегрированных

сред разработки (Integrated Development Environment, IDE), которые являются графическими интерфейсами, поддерживающими инструменты вроде текстовых редакторов, отладчиков, поиска по библиотеке и т. д.

IDLE

IDLE (http://bit.ly/py-idle) — это IDE, предназначенная только для Python, которая поставляется со стандартным дистрибутивом. Она основана на интерфейсе tkinter и имеет простой GUI.

PyCharm

PyCharm (http://www.jetbrains.com/pycharm/) — это относительно новая графическая IDE, имеющая множество возможностей. Версия для сообщества бесплатна, также вы можете получить бесплатную лицензию для профессиональной версии, чтобы использовать ее для обучения или работы над проектом с открытым исходным кодом. На рис. 12.1 показан ее начальный экран.

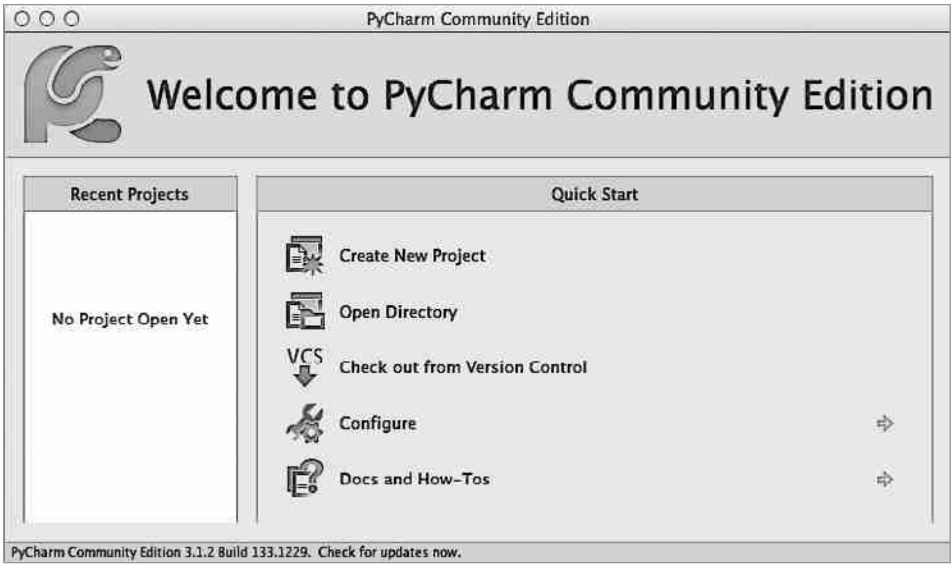


Рис. 12.1. Начальный экран PyCharm

IPython

iPython (http://ipython.org/), которую вы увидите в приложении В, — это платформа для публикации приложений, а также IDE с широкими возможностями.

Именуйте и документируйте

Вы не вспомните то, что написали. Иногда случается так, что я смотрю на код, даже на тот, который написал недавно, и не понимаю, откуда он взялся. Именно поэтому полезно документировать собственный код. Документация может включать в себя комментарии и строки документации, также полезно давать переменным, функциям, модулям и классам осмысленные имена. Однако не перегибайте палку, как в этом примере:

```
>>> # Здесь я собираюсь присвоить значение 10 переменной "num":
... num = 10
>>> # Надеюсь, это сработало
... print(num)
10
>>> # Фух.
```

Вместо этого напишите, почему вы присвоили значение `10`. Укажите, почему дали переменной именно имя `num`. Если вы пишете почтенный преобразователь температуры от шкалы Фаренгейта к шкале Цельсия, вам следует назвать переменные так, чтобы было понятно, что они делают, вместо того чтобы произвести на свет кучу волшебного кода. Небольшой тест также не повредит:

```
def ftoc(f_temp):
    "Convert Fahrenheit temperature <f_temp> to Celsius and return it."
    f_boil_temp = 212.0
    f_freeze_temp = 32.0
    c_boil_temp = 100.0
    c_freeze_temp = 0.0
    f_range = f_boil_temp – f_freeze_temp
    c_range = c_boil_temp – c_freeze_temp
    f_c_ratio = c_range / f_range
    c_temp = (f_temp – f_freeze_temp) * f_c_ratio + c_freeze_temp
    return c_temp
if __name__ == '__main__':
    for f_temp in [-40.0, 0.0, 32.0, 100.0, 212.0]:
        c_temp = ftoc(f_temp)
        print('%f F => %f C' % (f_temp, c_temp))
```

Запустим тесты:

```
$ python ftoc1.py
-40.000000 F => -40.000000 C
0.000000 F => -17.777778 C
32.000000 F => 0.000000 C
100.000000 F => 37.777778 C
212.000000 F => 100.000000 C
```

Мы можем сделать как минимум два улучшения.

- В языке Python нет констант, но таблица стилей PEP-8 рекомендует (http://bit.ly/pep-constant) использовать прописные буквы и подчеркивания (например, ALL_CAPS) при именовании переменных, которые должны считаться константами. Переименуем эти «константные» переменные в нашем примере.
- Поскольку мы заранее вычислили значения, основываясь на константах, перенесем их в верхнюю часть модуля. Таким образом, они будут рассчитываться только один раз при каждом вызове функции ftoc().

Так выглядит переделанный код:

```
F_BOIL_TEMP = 212.0
F_FREEZE_TEMP = 32.0
C_BOIL_TEMP = 100.0
C_FREEZE_TEMP = 0.0
F_RANGE = F_BOIL_TEMP - F_FREEZE_TEMP
C_RANGE = C_BOIL_TEMP - C_FREEZE_TEMP
F_C_RATIO = C_RANGE / F_RANGE
def ftoc(f_temp):
    "Convert Fahrenheit temperature <f_temp> to Celsius and return it."
    c_temp = (f_temp - F_FREEZE_TEMP) * F_C_RATIO + C_FREEZE_TEMP
    return c_temp
if __name__ == '__main__':
    for f_temp in [-40.0, 0.0, 32.0, 100.0, 212.0]:
        c_temp = ftoc(f_temp)
        print('%f F => %f C' % (f_temp, c_temp))
```

Тестируем код

Время от времени я делаю небольшое изменение в своем коде и говорю себе: «Выглядит неплохо, можно отправлять». А затем все ломается. Ой! Каждый раз, когда я делаю это (к счастью, со временем все реже и реже), я чувствую себя глупцом и клянусь, что в следующий раз напишу еще больше тестов.

Самый простой способ протестировать программы, написанные на Python, — добавить команды print(). Read-Evaluate-Print Loop (REPL) интерактивного интерпретатора позволяет вам быстро изменять код и тестировать изменения. Однако в производственном коде выражения print() использовать не стоит, поэтому вам нужно помнить о том, что следует удалять их. К тому же ошибки, связанные с копированием и вставкой, сделать очень легко.

pylint, pyflakes и PEP-8

Следующим шагом перед созданием настоящих программ для тестирования является использование контролера кода Python. Самыми популярными являются

pylint (http://www.pylint.org/) и pyflakes (http://bit.ly/pyflakes). Вы можете установить любой из них (или даже оба) с помощью pip:

```
$ pip install pylint
$ pip install pyflakes
```

Они проверяют на наличие реальных ошибок в коде (например, обращения к переменной до присвоения ей значения) и несоответствие стилю (как если бы код носил одновременно одежду в полоску и в клетку). Рассмотрим практически бессмысленную программу, в которой есть логическая и стилистическая ошибки:

```
a = 1
b = 2
print(a)
print(b)
print(c)
```

Так выглядит выходная информация от pylint:

```
$ pylint style1.py
No config file found, using default configuration
************* Module style1
C:  1,0: Missing docstring
C:  1,0: Invalid name "a" for type constant
  (should match (([A-Z_][A-Z0-9_]*)|(__.*__))$)
C:  2,0: Invalid name "b" for type constant
  (should match (([A-Z_][A-Z0-9_]*)|(__.*__))$)
E:  5,6: Undefined variable 'c'
```

Если пролистать дальше, к разделу Global evaluation, можно увидеть наш счет (10.0 — это высший балл):

```
Your code has been rated at -3.33/10
```

Ой! Сначала исправим ошибку. Строка вывода pylint, которая начинается с E, указывает на то, что найдена ошибка, которая заключается в том, что мы не присвоили значение переменной c до ее вывода на экран. Давайте это исправим:

```
a = 1
b = 2
c = 3
print(a)
print(b)
print(c)
$ pylint style2.py
No config file found, using default configuration
************* Module style2
C:  1,0: Missing docstring
C:  1,0: Invalid name "a" for type constant
  (should match (([A-Z_][A-Z0-9_]*)|(__.*__))$)
C:  2,0: Invalid name "b" for type constant
```

```
  (should match (([A-Z_][A-Z0-9_]*)|(__.*__))$)
C:  3,0: Invalid name "c" for type constant
  (should match (([A-Z_][A-Z0-9_]*)|(__.*__))$)
```

Отлично, больше строк, начинающихся с `E`, нет. Наш счет увеличился с `-3.33` до `4.29`:

```
Your code has been rated at 4.29/10
```

`pylint` хочет увидеть строку документации (короткий текстовый фрагмент в верхней части модуля или функции, описывающий код) и считает, что короткие имена переменных вроде `a`, `b` и `c` не очень аккуратны. Сделаем `pylint` счастливее, а наш код — еще лучше:

```
"Module docstring goes here"
def func():
    "Function docstring goes here. Hi, Mom!"
    first = 1
    second = 2
    third = 3
    print(first)
    print(second)
    print(third)
func()
$ pylint style3.py
No config file found, using default configuration
```

Жалоб нет. А какой у нас счет?

```
Your code has been rated at 10.00/10
```

Не так уж и плохо?

Еще одним контролером стиля является PEP-8 (https://pypi.python.org/pypi/pep8), вы можете установить его привычным способом:

```
$ pip install pep8
```

Что он скажет о нашей последней версии кода?

```
$ pep8 style3.py
style3.py:3:1: E302 expected 2 blank lines, found 1
```

Чтобы сделать код еще более стильным, он рекомендует добавить пустую строку после начальной строки документации модуля.

unittest

Мы убедились, что больше не оскорбляем чувство стиля богов кода, поэтому теперь можно перейти к настоящим тестам логики вашей программы.

Хорошим тоном является написание и запуск тестовых программ до отправки кода в систему контроля исходного кода. Написание тестов поначалу может быть

утомительным, но они действительно помогают вам находить проблемы быстрее, особенно регрессионные тесты (суть которых заключается в том, чтобы сломать то, что раньше работало). Болезненный опыт учит всех разработчиков тому, что даже самое маленькое изменение, которое, по их заверениям, не затрагивает другие области приложения, на самом деле влияет на них. Если вы взглянете на качественные пакеты Python, то заметите, что они поставляются с набором тестов.

Стандартная библиотека содержит не один, а целых два пакета для тестирования приложений. Начнем с `unittest` (https://docs.python.org/3/library/unittest.html). Мы напишем модуль, который записывает слова с прописной буквы. Наша первая версия будет использовать стандартную строковую функцию `capitalize()`, что, как вы увидите, приведет к неожиданным результатам. Сохраните этот файл под именем `cap.py`:

```
def just_do_it(text):
    return text.capitalize()
```

Основная идея тестирования заключается в том, чтобы понять, какой результат вы хотите получить при определенных входных данных (в нашем примере вы хотите получить текст, который ввели, записанным с прописной буквы), отправить результат функции тестирования, а затем проверить, получен ли ожидаемый результат. Ожидаемый результат называется утверждением (assertion), поэтому в рамках пакета `unittest` вы проверяете результат с помощью методов, чьи имена начинаются со слова `assert`, например `assertEqual`, показанного в следующем примере.

Сохраните этот сценарий тестирования под именем `test_cap.py`:

```
import unittest
import cap
class TestCap(unittest.TestCase):
    def setUp(self):
        pass
    def tearDown(self):
        pass
    def test_one_word(self):
        text = 'duck'
        result = cap.just_do_it(text)
        self.assertEqual(result, 'Duck')
    def test_multiple_words(self):
        text = 'a veritable flock of ducks'
        result = cap.just_do_it(text)
        self.assertEqual(result, 'A Veritable Flock Of Ducks')
if __name__ == '__main__':
    unittest.main()
```

Перед каждым методом тестирования вызывается метод `setUp()`, а после каждого из методов тестирования — метод `tearDown()`. Задачей этих методов является выделение и освобождение внешних ресурсов, необходимых для тестов, вроде

соединения с базой данных или создания некоторых тестовых данных. В нашем случае тесты автономны, и нам даже не нужно определять методы setUp() и tearDown(), однако создать их пустые версии не повредит. Сердцем наших тестов являются две функции с именами test_one_word() и test_multiple_words(). Каждая из них запускает определенную нами функцию just_do_it() с разными входными параметрами и проверяет, получен ли ожидаемый результат.

О'кей, запустим тест. Эта команда вызовет два наших метода тестирования:

```
$ python test_cap.py
 F.
 ======================================================================
 FAIL: test_multiple_words (__main__.TestCap)
 ----------------------------------------------------------------------
 Traceback (most recent call last):
   File "test_cap.py", line 20, in test_multiple_words
  self.assertEqual(result, 'A Veritable Flock Of Ducks')
 AssertionError: 'A veritable flock of ducks' != 'A Veritable Flock Of Ducks'
 – A veritable flock of ducks
 ?   ^         ^     ^  ^
 + A Veritable Flock Of Ducks
 ?   ^         ^     ^  ^
 ----------------------------------------------------------------------
 Ran 2 tests in 0.001s
 FAILED (failures=1)
```

Пакет устроил результат первой проверки (test_one_word), но не результат второй (test_multiple_words). Стрелки вверх (^) показывают, какие строки отличаются.

Что такого особенного в примере с несколькими словами? После прочтения документации для строковой функции capitalize (https://docs.python.org/3/library/stdtypes.html#str.capitalize) мы поняли причину проблемы: она увеличивает только первую букву первого слова. Возможно, нам сразу нужно было начать с чтения документации.

Нам нужна другая функция. После прочтения той страницы мы нашли функцию title() (https://docs.python.org/3/library/stdtypes.html#str.title). Изменим файл cap.py так, чтобы в нем вместо функции capitalize() использовалась функция title():

```
def just_do_it(text):
    return text.title()
```

Повторите тесты и взгляните на результат:

```
$ python test_cap.py
 ..
 ----------------------------------------------------------------------
 Ran 2 tests in 0.000s
 OK
```

Все прошло отлично. Хотя на самом деле нет. Нам нужно добавить в файл test_cap.py как минимум еще один метод:

```
    def test_words_with_apostrophes(self):
        text = "I'm fresh out of ideas"
        result = cap.just_do_it(text)
        self.assertEqual(result, "I'm Fresh Out Of Ideas")
```

Запустите тесты еще раз:

```
$ python test_cap.py
 ..F
 ======================================================================
 FAIL: test_words_with_apostrophes (__main__.TestCap)
 ----------------------------------------------------------------------
 Traceback (most recent call last):
   File "test_cap.py", line 25, in test_words_with_apostrophes
     self.assertEqual(result, "I'm Fresh Out Of Ideas")
 AssertionError: "I'M Fresh Out Of Ideas" != "I'm Fresh Out Of Ideas"
 – I'M Fresh Out Of Ideas
 ?   ^
 + I'm Fresh Out Of Ideas
 ?   ^
 ----------------------------------------------------------------------
 Ran 3 tests in 0.001s
 FAILED (failures=1)
```

Наша функция увеличила букву m в конструкции I'm. В документации к функции title() мы обнаружили, что она плохо работает с апострофами. Нам действительно стоило сначала прочитать ее текст целиком.

В самом конце документации стандартной библиотеки, касающейся строк, мы находим еще одного кандидата — вспомогательную функцию с именем capwords(). Используем ее в файле cap.py:

```
def just_do_it(text):
    from string import capwords
    return capwords(text)
$ python test_cap.py
 ...
 ----------------------------------------------------------------------
 Ran 3 tests in 0.004s
 OK
```

Наконец-то мы это сделали! Э-э-э, на самом деле нет. Нужно добавить еще один тест в файл test_cap.py:

```
    def test_words_with_quotes(self):
        text = "\"You're despicable,\" said Daffy Duck"
        result = cap.just_do_it(text)
        self.assertEqual(result, "\"You're Despicable,\" Said Daffy Duck")
```

Сработало?

```
$ python test_cap.py
 ...F
 ======================================================================
 FAIL: test_words_with_quotes (__main__.TestCap)
 ----------------------------------------------------------------------
 Traceback (most recent call last):
   File "test_cap.py", line 30, in test_words_with_quotes
     self.assertEqual(result, "\"You're
     Despicable,\" Said Daffy Duck")
 AssertionError: '"you\'re Despicable," Said Daffy Duck'
  != '"You\'re Despicable," Said Daffy Duck'
 - "you're Despicable," Said Daffy Duck
 ?  ^
 + "You're Despicable," Said Daffy Duck
 ?  ^
 ----------------------------------------------------------------------
 Ran 4 tests in 0.004s
 FAILED (failures=1)
```

Выглядит так, будто первая двойная кавычка смутила даже функцию `capwords`, нашего текущего фаворита. Она попробовала увеличить символ " и уменьшить все остальное (`You're`). Нам также нужно проверить, оставила ли функция-увеличитель остальную часть строки нетронутой.

Люди, которые зарабатывают тестированием на жизнь, способны замечать такие крайние случаи, но разработчики часто забывают о таких ситуациях, когда речь идет об их собственном коде.

Пакет `unittest` предоставляет небольшой, но мощный набор операторов, позволяющих вам проверять значения, убеждаться в том, что вы имеете необходимый класс, определять, сгенерировалась ли ошибка, и т. д.

Пакет doctest

Вторым пакетом для тестирования стандартной библиотеки является `doctest` (http://bit.ly/py-doctest). С помощью этого пакета вы можете писать тесты внутри строки документации, которые и сами будут служить документацией. Он выглядит как интерактивный интерпретатор: символы >>>, за которыми следует вызов, а затем результаты в следующей строке. Вы можете запустить некоторые тесты в интерактивном интерпретаторе и просто вставить результат в свой тестовый файл. Мы модифицируем файл `cap.py` (убрав тот проблемный тест с кавычками):

```
def just_do_it(text):
    """
    >>> just_do_it('duck')
    'Duck'
```

```
    >>> just_do_it('a veritable flock of ducks')
    'A Veritable Flock Of Ducks'
    >>> just_do_it("I'm fresh out of ideas")
    "I'm Fresh Out Of Ideas"
    """
    from string import capwords
    return capwords(text)
if __name__ == '__main__':
    import doctest
    doctest.testmod()
```

Когда вы его запустите, в случае успеха он не выведет ничего:

```
$ python cap.py
```

Запустите его с опцией `-v` (`verbose`, `verbose`), чтобы увидеть, что произошло на самом деле:

```
$ python cap.py -v
 Trying:
     just_do_it('duck')
 Expecting:
     'Duck'
 ok
 Trying:
     just_do_it('a veritable flock of ducks')
 Expecting:
     'A Veritable Flock Of Ducks'
 ok
 Trying:
     just_do_it("I'm fresh out of ideas")
 Expecting:
     "I'm Fresh Out Of Ideas"
 ok
 1 items had no tests:
     __main__
 1 items passed all tests:
    3 tests in __main__.just_do_it
 3 tests in 2 items.
3 passed and 0 failed.
 Test passed.
```

Пакет nose

Сторонний пакет `nose` (**https://nose.readthedocs.org/en/latest/**) — это еще одна альтернатива пакету `unittest`. Команда, позволяющая установить его, выглядит так:

```
$ pip install nose
```

Вам не нужно создавать класс, который содержит тестовые методы, как мы делали при работе с unittest. Любая функция, содержащая в своем имени слово test, будет запущена. Модифицируем нашего последнего тестировщика Unittest и сохраним его под именем test_cap_nose.py:

```
import cap
from nose.tools import eq_
def test_one_word():
    text = 'duck'
    result = cap.just_do_it(text)
    eq_(result, 'Duck')
def test_multiple_words():
    text = 'a veritable flock of ducks'
    result = cap.just_do_it(text)
    eq_(result, 'A Veritable Flock Of Ducks')
def test_words_with_apostrophes():
    text = "I'm fresh out of ideas"
    result = cap.just_do_it(text)
    eq_(result, "I'm Fresh Out Of Ideas")
def test_words_with_quotes():
    text = "\"You're despicable,\" said Daffy Duck"
    result = cap.just_do_it(text)
    eq_(result, "\"You're Despicable,\" Said Daffy Duck")
```

Запустим тесты:

```
$ nosetests test_cap_nose.py
 ...F
 ======================================================================
 FAIL: test_cap_nose.test_words_with_quotes
 ----------------------------------------------------------------------
 Traceback (most recent call last):
   File "/Users/.../site-packages/nose/case.py", line 198, in runTest
     self.test(*self.arg)
   File "/Users/.../book/test_cap_nose.py", line 23, in test_words_with_quotes
     eq_(result, "\"You're Despicable,\" Said Daffy Duck")
 AssertionError: '"you\'re Despicable," Said Daffy Duck'
     != '"You\'re Despicable," Said Daffy Duck'
----------------------------------------------------------------------
 Ran 4 tests in 0.005s
 FAILED (failures=1)
```

Мы нашли ту же ошибку, что и при использовании unittest, к счастью, в конце этой главы приводится упражнение, в котором вам предстоит ее исправить.

Другие фреймворки для тестирования

По некоторой причине людям нравится писать тестовые фреймворки для Python. Если вам любопытно, можете взглянуть на другие популярные решения вроде tox и py.test.

Постоянная интеграция

Когда ваша группа генерирует много кода каждый день, полезно автоматизировать тесты по мере появления изменений. Вы можете автоматизировать системы контроля версий так, чтобы тесты запускались по мере появления нового кода. Таким образом, каждый будет знать, что кто-то сломал билд и убежал обедать пораньше.

Эти системы велики, и я не буду рассматривать детали их установки и использования. Если они вам когда-нибудь понадобятся, вы будете знать, где их искать.

- buildbot (http://buildbot.net/). Эта система контроля версий, написанная на Python, автоматизирует построение, тестирование и выпуск кода.
- jenkins (http://jenkins-ci.org/). Система написана на Java, она выглядит наиболее предпочтительным инструментом для постоянной интеграции в данный момент.
- travis-ci (http://travis-ci.com/). Эта система автоматизирует проекты, размещенные на GitHub, она бесплатна для проектов с открытым исходным кодом.

Отлаживаем свой код

> Отладка кода вдвое сложнее, чем его написание. Так что если вы пишете код настолько умно, насколько можете, то вы по определению недостаточно сообразительны, чтобы его отлаживать.
>
> *Брайан Керниган (Brian Kernighan)*

Всегда тестируйте свой код. Чем лучше тесты, тем меньше вам предстоит работы в дальнейшем. Однако ошибки случаются, и их нужно исправлять. Самый простой способ выполнять отладку в Python — построчно выполнять код. Полезно отображать результат работы функции `vars()`, которая извлекает значения ваших локальных переменных, включая аргументы функций:

```
>>> def func(*args, **kwargs):
...     print(vars())
...
>>> func(1, 2, 3)
{'args': (1, 2, 3), 'kwargs': {}}
>>> func(['a', 'b', 'argh'])
{'args': (['a', 'b', 'argh'],), 'kwargs': {}}
```

Как вы уже знаете из раздела «Декораторы» главы 4, декоратор может вызывать код, располагающийся до или после функции, не модифицируя код внутри самой функции. Это значит, что вы можете использовать декоратор, чтобы выполнить какое-либо действие до или после вызова любой функции, а не только тех, которые написали вы. Определим декоратор `dump`, который позволяет вывести на экран

входные аргументы и выводимые значения любой функции по мере ее вызова (дизайнеры знают, что выходные данные нужно декорировать):

```
def dump(func):
    "Print input arguments and output value(s)"
    def wrapped(*args, **kwargs):
        print("Function name: %s" % func.__name__)
        print("Input arguments: %s" % ' '.join(map(str, args)))
        print("Input keyword arguments: %s" % kwargs.items())
        output = func(*args, **kwargs)
        print("Output:", output)
        return output
    return wrapped
```

Перейдем к декорируемой части. Это функция с именем `double()`, которая принимает именованные или безымянные числовые аргументы и возвращает их удвоенные значения в списке:

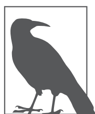
```
from dump1 import dump
@dump
def double(*args, **kwargs):
    "Double every argument"
    output_list = [ 2 * arg for arg in args ]
    output_dict =  { k:2*v for k,v in kwargs.items() }
    return output_list, output_dict
if __name__ == '__main__':
    output = double(3, 5, first=100, next=98.6, last=-40)
```

Запустите пример:

```
$ python test_dump.py
Function name: double
Input arguments: 3 5
Input keyword arguments: dict_items([('last', -40), ('first', 100),
 ('next', 98.6)])
Output: ([6, 10], {'last': -80, 'first': 200, 'next': 197.2})
```

Отлаживаем с помощью pdb

Эти приемы полезны, но иногда ничто не сможет заменить настоящий отладчик. Большинство IDE содержат отладчики, чьи возможности и пользовательские интерфейсы могут варьироваться. В этом разделе я опишу использование стандартного отладчика Python `pdb` (https://docs.python.org/3/library/pdb.html).



Если вы запускаете программу с флагом –i, при ее неудачном завершении Python вернет вас в интерактивный интерпретатор.

Рассмотрим программу с ошибкой, которая зависит от входных данных, — такую ошибку может быть особенно трудно найти. Это реальная ошибка, возникшая в ранние дни программирования, она довольно долго сбивала с толку программистов.

Мы собираемся считать файл, содержащий названия стран и их столиц, разделенные запятыми, и вывести их на экран в формате «столица, страна». Учтите, что прописные буквы в словах могут быть неправильно расставлены, поэтому нам нужно исправить это при выводе на экран. В файле также могут быть лишние пробелы, вам нужно избавиться и от них. Наконец, несмотря на то что было бы логично считать весь файл до конца, по какой-то причине наш менеджер сказал нам остановиться, если мы встретим слово `quit` (состоящее из смеси прописных и строчных букв). Так выглядит файл с данными:

```
France, Paris
venuzuela,caracas
  LithuniA,vilnius
     quit
```

Разработаем алгоритм (способ решения задачи). Это псевдокод, он выглядит как программа, но является лишь способом выразить логику простым языком до преобразования его в настоящую программу. Одна из причин, по которым программисты любят Python, — он выглядит очень похожим на псевдокод, поэтому его не так трудно преобразовать в рабочую программу:

```
для каждой строки в текстовом файле
считать строку
удалить пробелы в начале и конце строки
если найдена строка "quit" в строке, записанной в нижнем регистре
остановиться
иначе
разделить страну и столицу символом запятой
удалить пробелы в начале и конце
записать страну и столицу с прописной буквы
вывести на экран столицу, запятую и страну
```

Нам нужно удалить из имен начальные и конечные пробелы, поскольку это является требованием к программе. Аналогично мы поступаем со сравнением со строкой `quit` и записью названий страны и города с прописной буквы. Имея это в виду, напишем файл `capitals.py`, который точно будет работать корректно:

```
def process_cities(filename):
    with open(filename, 'rt') as file:
        for line in file:
            line = line.strip()
            if 'quit' in line.lower():
                return
            country, city = line.split(',')
```

```
            city = city.strip()
            country = country.strip()
            print(city.title(), country.title(), sep=',')
if __name__ == '__main__':
    import sys
    process_cities(sys.argv[1])
```

Протестируем программу с помощью файла, созданного ранее. На старт, внимание, марш:

```
$ python capitals.py  cities1.csv
Paris,France
Caracas,Venuzuela
Vilnius,Lithunia
```

Выглядит отлично! Программа прошла один тест, поэтому отправим ее на производство обрабатывать столицы и страны со всего мира, пока она не ошибется на данном файле:

```
argentina,buenos aires
bolivia,la paz
brazil,brasilia
chile,santiago
colombia,Bogotá
ecuador,quito
falkland islands,stanley
french guiana,cayenne
guyana,georgetown
paraguay,Asunción
peru,lima
suriname,paramaribo
uruguay,montevideo
venezuela,caracas
quit
```

Программа завершается после вывода всего пяти строк, несмотря на то что в файле их было 15, как показано здесь:

```
$ python capitals.py  cities2.csv
Buenos Aires,Argentina
La Paz,Bolivia
Brazilia,Brazil
Santiago,Chile
Bogotá,Colombia
```

Что случилось? Мы можем продолжать редактировать файл `capitals.py`, размещая выражения `print()` в тех местах, где может возникнуть ошибка, но посмотрим, сможет ли нам помочь отладчик.

Для того чтобы использовать отладчик, импортируйте модуль pdb из командной строки, введя -m pdb, например, так:

```
$ python -m pdb capitals.py cities2.csv
> /Users/williamlubanovic/book/capitals.py(1)<module>()
-> def process_cities(filename):
(Pdb)
```

Это запустит программу и разместит вас на первой строке. Если вы введете символ c (от слова continue — «продолжить»), программа будет работать, пока не завершится либо естественным образом, либо из-за ошибки:

```
 (Pdb) c
Buenos Aires,Argentina
La Paz,Bolivia
Brazilia,Brazil
Santiago,Chile
Bogotá,Colombia
The program finished and will be restarted
> /Users/williamlubanovic/book/capitals.py(1)<module>()
-> def process_cities(filename):
```

Программа завершилась нормально, точно так же, как и раньше, когда мы запускали ее вне отладчика. Попробуем запустить ее снова, использовав специальные команды, чтобы сузить место поиска проблемы. Похоже, имеет место логическая ошибка, а не синтаксическая проблема или исключение (из-за них мы бы увидели сообщение об ошибке).

Введите s (step — «шаг»), чтобы пройти по отдельным строкам кода. Это позволит пройти по всем строкам — вашим, стандартной библиотеки и любых других используемых вами модулей. Когда вы применяете команду s, вы также входите во все функции и проходите каждую построчно. Введите n (next — «следующий»), чтобы идти по шагам, но не заходить внутрь функций: когда вы находитесь на строке, где вызывается функция, эта команда выполняет всю функцию и вы оказываетесь на следующей строке. Используйте s, если вы не уверены в том, где конкретно есть проблема, а n — если уверены, что некоторая функция проблем не вызывает, особенно если это длинная функция. Зачастую вы будете проходить построчно весь свой код и пропускать библиотечный, поскольку подразумевается, что он хорошо протестирован. Мы используем s, чтобы начать двигаться от начала программы к функции process_cities():

```
 (Pdb) s
 > /Users/williamlubanovic/book/capitals.py(12)<module>()
 -> if __name__ == '__main__':
(Pdb) s
 > /Users/williamlubanovic/book/capitals.py(13)<module>()
 -> import sys
```

```
(Pdb) s
 > /Users/williamlubanovic/book/capitals.py(14)<module>()
 -> process_cities(sys.argv[1])
(Pdb) s
 --Call--
 > /Users/williamlubanovic/book/capitals.py(1)process_cities()
 -> def process_cities(filename):
 (Pdb) s
 > /Users/williamlubanovic/book/capitals.py(2)process_cities()
 -> with open(filename, 'rt') as file:
```

Введите l (list — «перечислить»), чтобы увидеть следующие несколько строк своей программы

```
 (Pdb) l
  1      def process_cities(filename):
  2  ->     with open(filename, 'rt') as file:
  3             for line in file:
  4                 line = line.strip()
  5                 if 'quit' in line.lower():
  6                     return
  7                 country, city = line.split(',')
  8                 city = city.strip()
  9                 country = country.strip()
 10                 print(city.title(), country.title(), sep=',')
 11
(Pdb)
```

Стрелка (->) указывает на текущую строку.

Мы могли бы и дальше применять команды s или n в надежде что-то найти, но давайте использовать одну из главных особенностей отладчика — точки останова. Точка останова останавливает выполнение программы на указанной вами строке. В данном случае мы хотим узнать, почему функция process_cities() вызывает завершение программы до прочтения всех введенных строк. Строка 3 (for line in file:) будет считывать каждую строку входного файла, поэтому она выглядит невинно. Единственное место, где мы можем вернуться из функции до прочтения всех данных, — это строка 6 (return). Поставим точку останова на строке 6:

```
 (Pdb) b 6
Breakpoint 1 at /Users/williamlubanovic/book/capitals.py:6
```

Далее продолжим выполнение программы до тех пор, пока она либо не достигнет точки останова, либо не завершится обычным образом:

```
 (Pdb) c
 Buenos Aires,Argentina
 La Paz,Bolivia
 Brasilia,Brazil
 Santiago,Chile
```

```
 Bogotá,Colombia
 > /Users/williamlubanovic/book/capitals.py(6)process_cities()
 -> return
```

Ага, она остановилась на точке останова в строке 6. Это показывает, что программа хочет завершиться после прочтения страны, которая идет вслед за Колумбией. Выведем значение переменной line, чтобы увидеть, что мы только что считали:

```
(Pdb) p line
'ecuador,quito'
```

Что такого особенного... а, забудьте.

Серьезно? Столица называется *quit*o? Наш менеджер не ожидал, что строка quit станет частью входных данных, поэтому показалось логичным использовать ее в качестве контрольного значения (индикатора конца). Вам следует отправиться прямо к нему и сказать все как на духу, я подожду.

Если после этого у вас все еще есть работа, можете просмотреть все точки останова с помощью команды b:

```
 (Pdb) b
Num Type         Disp Enb   Where
1   breakpoint   keep yes   at /Users/williamlubanovic/book/capitals.py:6
    breakpoint already hit 1 time
```

Команда l покажет вам строки кода, текущую строку (->) и все имеющиеся точки останова (B). Вызов команды l без аргументов выведет все строки, начиная с точки предыдущего вызова команды l, поэтому включите в вызов опциональный параметр — стартовую строку (в нашем примере начнем с 1):

```
 (Pdb) l 1
  1      def process_cities(filename):
  2         with open(filename, 'rt') as file:
  3             for line in file:
  4                 line = line.strip()
  5                 if 'quit' in line.lower():
  6 B->                 return
  7                 country, city = line.split(',')
  8                 city = city.strip()
  9                 country = country.strip()
 10                 print(city.title(), country.title(), sep=',')
 11
```

Теперь модифицируем наш тест так, чтобы выполнялась проверка на полное совпадение со строкой quit, без всяких других символов:

```
def process_cities(filename):
    with open(filename, 'rt') as file:
        for line in file:
```

```
            line = line.strip()
            if 'quit' == line.lower():
                return
            country, city = line.split(',')
            city = city.strip()
            country = country.strip()
            print(city.title(), country.title(), sep=',')
if __name__ == '__main__':
    import sys
    process_cities(sys.argv[1])
```

Запустим программу еще раз:

```
$ python capitals2.py cities2.csv
Buenos Aires,Argentina
La Paz,Bolivia
Brasilia,Brazil
Santiago,Chile
Bogotá,Colombia
Quito,Ecuador
Stanley,Falkland Islands
Cayenne,French Guiana
Georgetown,Guyana
Asunción,Paraguay
Lima,Peru
Paramaribo,Suriname
Montevideo,Uruguay
Caracas,Venezuela
```

Только что мы кратко рассмотрели отладчик — этого достаточно, чтобы показать вам, что вы можете сделать и какие команды будете использовать бóльшую часть времени.

Помните: больше тестов — меньше отладки.

Записываем в журнал сообщения об ошибках

В какой-то момент вам может понадобиться перейти от использования выражений `print()` к записи сообщений в журнал. Журнал, как правило, представляет собой системный файл, в котором накапливаются сообщения, содержащие полезную информацию вроде временной метки или имени пользователя, запустившего программу. Зачастую журналы ежедневно ротируются (переименовываются) и сжимаются, благодаря чему они не переполняют ваш диск и не создают проблем. Если у вашей программы что-то пошло не так, вы можете просмотреть соответствующий файл журнала, чтобы увидеть, что произошло. Содержимое исключений особенно полезно записывать в журнал, поскольку оно подсказывает

вам номер строки, после выполнения которой программа завершилась, и причину такого завершения.

Для журналирования используется модуль стандартной библиотеки logging (http://bit.ly/py-logging). Большинство его описаний я считаю немного непонятными. Спустя некоторое время эти описания будут казаться осмысленными, но в первый раз они выглядят чересчур сложными. Модуль logging содержит следующие концепции:

- сообщение, которое вы хотите сохранить в журнал;
- уровни приоритета и соответствующие функции — debug(), info(), warn(), error() и critical();
- один или несколько объектов журналирования для основной связи с модулем;
- обработчики, которые направляют значение в терминал, файл, базу данных или куда-нибудь еще;
- средства форматирования выходных данных;
- фильтры, которые принимают решения в зависимости от входных данных.

Рассмотрим простейший пример журналирования — просто импортируем модуль и воспользуемся некоторыми из его функций:

```
>>> import logging
>>> logging.debug("Looks like rain")
>>> logging.info("And hail")
>>> logging.warn("Did I hear thunder?")
WARNING:root:Did I hear thunder?
>>> logging.error("Was that lightning?")
ERROR:root:Was that lightning?
>>> logging.critical("Stop fencing and get inside!")
CRITICAL:root:Stop fencing and get inside!
```

Вы заметили, что вызовы debug() и info() не сделали ничего, а два других вывели на экран строку УРОВЕНЬ:root: перед каждым сообщением? Пока они выглядят как выражение print(), имеющее несколько личностей.

Но это полезно. Вы можете выполнить поиск определенного значения уровня в журнале, чтобы найти определенные сообщения, сравнить временны́е метки и увидеть, что случилось перед тем, как упал ваш сервер, и т. д.

Усиленные раскопки в документации отвечают на первую загадку (на вторую мы ответим уже через пару страниц): уровень приоритета по умолчанию — WARNING, он будет записан в журнал, когда мы вызовем первую функцию (logging.debug()). Мы можем указать уровень по умолчанию с помощью функции basicConfig(). Самый низкий уровень — DEBUG, это дает возможность поймать более высокие уровни:

```
>>> import logging
>>> logging.basicConfig(level=logging.DEBUG)
>>> logging.debug("It's raining again")
```

```
DEBUG:root:It's raining again
>>> logging.info("With hail the size of hailstones")
INFO:root:With hail the size of hailstones
```

Мы сделали это с помощью стандартных функций журналирования, не создавая специализированный объект. Каждый объект журналирования имеет имя. Создадим объект, который называется bunyan:

```
>>> import logging
>>> logging.basicConfig(level='DEBUG')
>>> logger = logging.getLogger('bunyan')
>>> logger.debug('Timber!')
DEBUG:bunyan:Timber!
```

Если имя объекта журналирования содержит точки, они разделяют уровни иерархии таких объектов, каждый из которых потенциально имеет разные приоритеты. Это означает, что объект с именем quark выше, чем объект с именем quark.charmed. На вершине иерархии находится корневой объект журналирования с именем ' '.

До этого момента мы только выводили сообщения, что практически не отличается от функции print().Чтобы направить сообщения в разные места назначения, используем обработчики. Самым распространенным местом назначения является файл журнала, направить туда сообщения можно так:

```
>>> import logging
>>> logging.basicConfig(level='DEBUG', filename='blue_ox.log')
>>> logger = logging.getLogger('bunyan')
>>> logger.debug("Where's my axe?")
>>> logger.warn("I need my axe")
>>>
```

Ага, строки больше не показываются на экране, вместо этого они попадают в файл blue_ox.log:

```
DEBUG:bunyan:Where's my axe?
WARNING:bunyan:I need my axe
```

Вызов функции basicConfig() и передача имени файла в качестве аргумента создали для вас объект типа FileHandler и сделали его доступным объекту журналирования. Модуль журналирования содержит как минимум 15 обработчиков для отправки сообщений в разные места вроде электронной почты, веб-серверов, экранов и файлов.

Наконец, вы можете управлять форматом сообщений журнала. В нашем первом примере мы использовали формат, применяемый по умолчанию, в результате чего появилась следующая строка:

```
WARNING:root:Message...
```

Если вы предоставите строку `format` функции `basicConfig()`, то можете изменить формат по собственному желанию:

```
>>> import logging
>>> fmt = '%(asctime)s %(levelname)s %(lineno)s %(message)s'
>>> logging.basicConfig(level='DEBUG', format=fmt)
>>> logger = logging.getLogger('bunyan')
>>> logger.error("Where's my other plaid shirt?")
2014-04-08 23:13:59,899 ERROR 1 Where's my other plaid shirt?
```

Мы позволили объекту журналирования снова отправить выходные данные на экран, но изменили их формат. Модуль logging распознал количество имен переменных в строке формата `fmt`. Мы использовали `asctime` (дата и время как строка ISO 8601), `levelname`, `lineno` (номер строки) и само сообщение в переменной `message`. Существуют и другие встроенные переменные, вы можете предоставить и свои собственные переменные.

Пакет logging содержит гораздо больше особенностей, чем можно описать в этом небольшом обзоре. Вы можете писать в несколько журналов одновременно, указывая разные приоритеты и форматы. Этот пакет довольно гибок, но иногда это достигается за счет простоты.

Оптимизируем код

Обычно Python работает довольно быстро, однако иногда его скорости не хватает. В большинстве случаев вы можете ускорить работу, выбрав более качественный алгоритм или структуру данных. Идея заключается в том, чтобы знать, где это сделать. Даже опытные программисты ошибаются довольно часто. Вам нужно быть очень осторожными и семь раз отмерить, прежде чем отрезать. Это приводит нас к использованию таймеров.

Измеряем время

Вы уже видели, что функция `time` модуля `time` возвращает текущее время в формате `epoch` как число секунд с плавающей точкой. Быстрый способ засечь время — получить текущее время, что-то сделать, получить новое время и вычесть из него первое. Напишем соответствующий код и назовем файл `time1.py`:

```
from time import time
t1 = time()
num = 5
num *= 2
print(time() - t1)
```

В этом примере мы измеряем время, которое требуется на присвоение значения 5 переменной num и умножение его на 2. Этот пример не является реалистичным тестом производительности, это лишь пример того, как замерить время выполнения произвольного кода. Попробуйте запустить его несколько раз, чтобы увидеть, что время может варьироваться:

```
$ python time1.py
2.1457672119140625e-06
$ python time1.py
2.1457672119140625e-06
$ python time1.py
2.1457672119140625e-06
$ python time1.py
1.9073486328125e-06
$ python time1.py
3.0994415283203125e-06
```

Программа работала две-три миллионные доли секунды. Попробуем выполнить что-то помедленнее, вроде функции sleep. Если мы усыпим выполнение на секунду, наш таймер покажет значение, чуть больше секунды. Сохраните файл под именем time2.py:

```
from time import time, sleep
t1 = time()
sleep(1.0)
print(time() – t1)
```

Чтобы быть уверенным в результатах, запустим программу несколько раз:

```
$ python time2.py
1.000797986984253
$ python time2.py
1.0010130405426025
$ python time2.py
1.0010390281677246
```

Как и ожидалось, программе для работы требуется около секунды. Если бы это оказалось не так, то либо нашему таймеру, либо функции sleep() должно было бы стать стыдно.

Существует более удобный способ измерить время выполнения фрагментов кода вроде этого — стандартный модуль timeit (http://bit.ly/py-timeit). У него имеется функция с именем, как вы уже догадались, timeit(), которая запустит ваш код заданное количество раз и выведет результаты. Ее синтаксис выглядит так: timeit.timeit(*код, число, количество_раз*).

В примерах этого раздела код должен находиться в кавычках, чтобы он выполнялся не после нажатия кнопки **Return**, а лишь внутри функции timeit(). (В следу-

ющем разделе вы увидите, как можно измерить время выполнения некоторой функции, передав ее имя в функцию `timeit()`.) Запустим предыдущий пример и измерим время его выполнения. Назовем этот файл `timeit1.py`:

```
from timeit import timeit
print(timeit('num = 5; num *= 2', number=1))
```

Запустим его несколько раз:

```
$ python timeit1.py
2.5600020308047533e-06
$ python timeit1.py
1.9020008039660752e-06
$ python timeit1.py
1.7380007193423808e-06
```

Опять же эти две строки кода выполняются примерно за две миллионные доли секунды. Мы можем использовать аргумент `repeat` функции `repeat()` модуля `timeit`, чтобы выполнить код большее количество раз. Сохраните этот файл под именем `timeit2.py`:

```
from timeit import repeat
print(repeat('num = 5; num *= 2', number=1, repeat=3))
Попробуйте запустить его, чтобы увидеть what transpires:
$ python timeit2.py
[1.691998477326706e-06, 4.070025170221925e-07, 2.4700057110749185e-07]
```

Первый запуск занял две миллионные доли секунды, а второй и третий прошли быстрее. Почему? Для этого может быть много причин. Это могло произойти, например, потому, что мы тестировали очень небольшой фрагмент кода и скорость его выполнения зависит от того, что компьютер делал в эти моменты, как система Python оптимизировала вычисления, и от многого другого.

Или же это могла быть случайность. Попробуем сделать что-то более реалистичное, чем присвоение переменных и вызов функции `sleep()`. Мы измерим производительность, сравнив эффективность нескольких алгоритмов (программной логики) и структур данных (механизмов хранения).

Алгоритмы и структуры данных

Дзен Python (http://bit.ly/zen-py) гласит: «Должен существовать один, и желательно только один, очевидный способ сделать это». К сожалению, иногда он не является очевидным и вам приходится сравнивать альтернативные варианты. Например, что лучше использовать для создания списка: цикл `for` или включение списка? И что на самом деле значит «лучше»: быстрее, проще для понимания, менее затратно по ресурсам или более характерно для Python?

В следующем упражнении мы создадим список разными способами, сравнив скорость, читаемость и стиль. Перед вами файл time_lists.py:

```
from timeit import timeit
def make_list_1():
    result = []
    for value in range(1000):
        result.append(value)
    return result
def make_list_2():
    result = [value for value in range(1000)]
    return result
print('make_list_1 takes', timeit(make_list_1, number=1000), 'seconds')
print('make_list_2 takes', timeit(make_list_2, number=1000), 'seconds')
```

В каждой функции мы добавляем в список 1000 элементов и вызываем каждую функцию 1000 раз. Обратите внимание на то, что в этом тесте мы вызываем функцию timeit(), передавая ей имя функции в качестве первого аргумента вместо кода. Давайте ее запустим:

```
$ python time_lists.py
make_list_1 takes 0.14117428699682932 seconds
make_list_2 takes 0.06174145900149597 seconds
```

Включение списка отработало как минимум в два раза быстрее, чем добавление элементов в список с помощью функции append(). Как правило, включение быстрее, чем создание вручную.

Используйте эти идеи, чтобы сделать свой код быстрее.

Cython, NumPy и расширения C

Если вы усердно работаете, но все еще не можете достичь необходимой производительности, у вас есть и другие варианты.

Cython (http://cython.org/) — это гибрид языков Python и C, разработанный для преобразования Python: в скомпилированный код языка C внесены некоторые улучшения производительности. Эти аннотации относительно малы, они похожи на объявление типов некоторых переменных, аргументов функций или возвращаемых функциями значений. Подобные подсказки сделают научные вычисления, выполняющиеся в циклах, гораздо быстрее — в 1000 раз. Документацию и примеры можете найти в Cython wiki (https://github.com/cython/cython/wiki).

Из приложения В вы можете подробнее узнать о NumPy. Это математическая библиотека Python, написанная для ускорения на C.

Многие части Python и его стандартной библиотеки написаны на C для скорости и обернуты кодом на Python для удобства. При написании приложений эти приемы доступны и вам. Если вы знаете C и Python и действительно хотите, чтобы ваш код «летал», напишите расширение на языке C — это труднее, но улучшение оправдает затраченные усилия.

PyPy

Около 20 лет назад, когда язык Java только появился, он был медленным, как шнауцер, больной артритом. Но когда он стал дорого стоить компании Sun и прочим, они вложили миллионы в оптимизацию интерпретатора Java и лежащей в его основе виртуальной машины Java (Java Virtual Machine, JVM), заимствуя приемы из уже существовавших тогда языков Smalltalk и LISP. Компания Microsoft также вложила много усилий в оптимизацию своего языка C# и .NET VM.

Языком Python никто не владеет, поэтому никто так сильно не старается сделать его быстрее. Вы, возможно, используете стандартную реализацию Python. Она написана на C и часто называется CPython (не путать с Cython).

Как и языки PHP, Perl и даже Java, Python не компилируется в машинный код, он преобразуется в промежуточный язык (он называется байт-кодом или p-кодом), который затем интерпретирует виртуальная машина.

PyPy (http://pypy.org/) — это новый интерпретатор Python, который пользуется некоторыми приемами, ускорившими язык программирования Java. Тесты производительности интерпретатора (http://speed.pypy.org/) показывают, что PyPy в каждом тесте быстрее CPython в среднем в шесть раз и до 20 раз в отдельных случаях. Он работает с Python 2 и 3. Вы можете загрузить его и использовать вместо CPython. PyPy постоянно улучшается и однажды может заменить CPython. Чтобы узнать, подходит ли он вам, посетите его официальный сайт.

Управление исходным кодом

Когда вы работаете над небольшой группой программ, то обычно можете отслеживать внесенные собственноручно изменения — до тех пор, пока не сделаете глупую ошибку и не потеряете несколько дней работы. Системы управления исходным кодом защитят ваш код от сил зла в лице вас самих. Если вы работаете в группе, управление исходным кодом становится необходимостью. Для этой области было создано множество коммерческих и бесплатных решений. Наиболее популярными в мире открытого исходного кода (где и живет Python) являются Mercurial и Git. Они оба являются примерами распределенных систем контроля версий, которые создают несколько копий репозиториев кода. Ранние системы вроде Subversion работают на одном сервере.

Mercurial

Mercurial (http://mercurial.selenic.com/) написан на Python. Научиться пользоваться им довольно легко, он имеет множество подкоманд для загрузки кода из репозитория Mercurial, добавления файлов, проверки на наличие изменений и объединения изменений из разных источников. bitbucket (https://bitbucket.org/) и другие сайты (http://bit.ly/merc-host) предлагают бесплатный или коммерческий хостинг.

Git

Git (http://git-scm.com/) изначально создавался для разработки ядра Linux, но теперь является доминирующим в области открытого исходного кода в целом. Он похож на Mercurial, хотя некоторые считают, что обучиться ему сложнее. GitHub (http://github.com/) — это самый крупный хостинг для git, содержащий более миллиона репозиториев, но существует и множество других хостов (http://bit.ly/githost-scm).

Отдельные примеры программ из этой книги доступны в публичном репозитории git на GitHub (https://github.com/madscheme/introducing-python). Если у вас установлена программа git, вы можете загрузить их с помощью следующей команды:

```
$ git clone https://github.com/madscheme/introducing-python
```

Вы также можете загрузить код, нажав на следующие кнопки на странице GitHub:

- Clone in Desktop (Клонировать на Рабочий стол), чтобы открыть версию git, установленную на ваш компьютер;
- Download ZIP (Загрузить архив), чтобы получить архивированную версию программ.

Если у вас нет git, но вы хотите попробовать с ним поработать, прочтите инструкцию по установке (http://bit.ly/git-install). Здесь я буду говорить о версии с командной строкой, но вам могут быть интересны сайты вроде GitHub, предоставляющие дополнительные услуги, которые в некоторых случаях использовать было бы проще: git имеет много возможностей, но не всегда интуитивно понятен.

Проведем тест-драйв. Далеко уходить не будем, просто посмотрим, как работают некоторые команды.

Создадим новую папку и перейдем в нее:

```
$ mkdir newdir
$ cd newdir
```

Создадим локальный репозиторий git в текущей папке `newdir`:

```
$ git init
Initialized empty Git repository in /Users/williamlubanovic/newdir/.git/
```

Создадим в папке `newdir` файл с кодом, который называется `test.py`, содержащий следующее:

```
print('Oops')
```

Добавим файл в репозиторий git:

```
$ git add test.py
```

Что вы об этом думаете, мистер git?

```
$ git status
On branch master
```

```
Initial commit
Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
    new file:   test.py
```

Это значит, что файл `test.py` стал частью локального репозитория, но изменения еще не были отправлены. Исправим это:

```
$ git commit -m "simple print program"
 [master (root-commit) 52d60d7] my first commit
  1 file changed, 1 insertion(+)
  create mode 100644 test.py
```

Строка `-m "my first commit"` является вашим комментарием. Если вы ее опустите, git выведет на экран редактор и тем самым предложит вам ввести сообщение. Оно становится частью истории изменений нашего файла.

Взглянем на текущий статус:

```
$ git status
 On branch master
 nothing to commit, working directory clean
```

О'кей, все текущие изменения были отправлены. Это значит, что мы можем менять содержимое файла и не беспокоиться о том, что потеряем его оригинал. Внесем изменение в файл `test.py` — заменим `Oops` на `Ops!` и сохраним файл:

```
print('Ops!')
```

Посмотрим, что теперь думает git:

```
$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -<file>..." to discard changes in working directory)
    modified:   test.py
no changes added to commit (use "git add" and/or "git commit -a")
```

Используйте команду `git diff`, чтобы увидеть, какие строки изменились с момента последней отправки:

```
$ git diff
 diff --git a/test.py b/test.py
 index 76b8c39..62782b2 100644
 --a/test.py
 +++ b/test.py
 @@ -1 +1 @@
 -print('Oops')
 +print('Ops!')
```

Если вы попробуете отправить это изменение сейчас, git пожалуется:

```
$ git commit -m "change the print string"
On branch master
Changes not staged for commit:
    modified:   test.py
no changes added to commit
```

Фраза staged for commit означает, что вам нужно добавить файл, что в примерном переводе выглядит как «Эй, git, смотри сюда!»:

```
$ git add test.py
```

Вы также могли ввести команду git add, чтобы добавить все измененные файлы в текущий каталог, — это удобно, когда вы изменили несколько файлов, чтобы гарантировать, что отправите все изменения. Теперь мы можем отправить изменения:

```
$ git commit -m "my first change"
 [master e1e11ec] my first change
  1 file changed, 1 insertion(+), 1 deletion(-)
```

Если вы хотите увидеть все те ужасные вещи, которые проделывали с файлом test.py, начиная с недавних, используйте команду git log:

```
$ git log test.py
 commit e1e11ecf802ae1a78debe6193c552dcd15ca160a
 Author: William Lubanovic <bill@madscheme.com>
 Date:   Tue May 13 23:34:59 2014 -0500
     change the print string
 commit 52d60d76594a62299f6fd561b2446c8b1227cfe1
 Author: William Lubanovic <bill@madscheme.com>
 Date:   Tue May 13 23:26:14 2014 -0500
     simple print program
```

Клонируйте эту книгу

Вы можете получить копию всех программ этой книги. Посетите репозиторий git (https://github.com/madscheme/introducing-python) и следуйте инструкциям по их копированию на ваш локальный компьютер. Если у вас есть git, запустите команду git clone https://github.com/madscheme/introducing-python, чтобы создать репозиторий git на вашем компьютере. Вы также можете загрузить файлы в формате ZIP.

Как узнать больше

Вы прочитали лишь введение. Скорее всего, в нем говорится слишком много о том, что вам не нужно, и слишком мало о том, что вам интересно. Позвольте мне порекомендовать некоторые ресурсы, связанные с Python, которые я считаю полезными.

Книги

Я обнаружил, что книги, представленные в следующем списке, особенно полезны. Их уровень различается от начального до продвинутого, в них описываются и Python 2, и Python 3.

- *Barry P.* Head First Python. — O'Reilly, 2010.
- *Beazley D. M.* Python Essential Reference. 4th ed. — Addison-Wesley, 2009.
- *Beazley D M., Jones B. K.* Python Cookbook. 3rd ed. — O'Reilly, 2013.
- *Chun W.* Core Python Applications Programming. 3rd ed. — Prentice Hall, 2012.
- *McKinney W.* Python for Data Analysis: Data Wrangling with Pandas, NumPy, and IPython. — O'Reilly, 2012.
- *Summerfield M.* Python in Practice: Create Better Programs Using Concurrency, Libraries, and Patterns. — Addison-Wesley, 2013.

Конечно же, хороших книг гораздо больше (https://wiki.python.org/moin/PythonBooks).

Сайты

Вот несколько сайтов, где вы можете найти полезные руководства:

- Learn Python the Hard Way, автор Зед Шоу (Zed Shaw) (http://learnpythonthehardway.org/book/);
- Dive Into Python 3, автор Марк Пилгрим (Mark Pilgrim) (http://www.diveintopython3.net/);
- Mouse Vs. Python, автор Майкл Дрисколл (Michael Driscoll) (http://www.blog.pythonlibrary.org/).

Если вам интересно узнавать о том, что происходит в мире Python, обратите внимание на эти новостные сайты:

- http://bit.ly/comp-lang-python;
- http://bit.ly/comp-lang-py-announce;
- http://www.reddit.com/r/python;
- Planet Python (http://planet.python.org/).

Наконец, рассмотрим сайты, с которых можно скачать хороший код:

- The Python Package Index (https://pypi.python.org/pypi);
- Stackoverflow Python Questions (http://stackoverflow.com/questions/tagged/python);
- ActiveState Python recipes (http://code.activestate.com/recipes/langs/python/);
- Python packages trending on GitHub (https://github.com/trending?l=python).

Группы

В сообществах программистов вы можете найти множество типажей: энтузиастов, спорщиков, глупцов, хипстеров, интеллигентов и множество других. Сообщество Python довольно дружелюбно. Вы можете найти группы, увлекающиеся Python, в зависимости от вашего местонахождения. Проводят встречи и местные пользовательские группы по всему миру (https://wiki.python.org/moin/LocalUserGroups). Другие группы распределены по всему миру и основываются на общих интересах. Например, PyLadies (http://www.pyladies.com/) — это сеть женщин, заинтересованных в Python и ПО с открытым исходным кодом.

Конференции

Самые крупные из множества конференций (http://www.pycon.org/) и совещаний по всему миру (https://www.python.org/community/workshops/) проводятся в Северной Америке (https://us.pycon.org/) и Европе (https://europython.eu/en/).

Coming Attractions

Но погодите, это еще не конец! В приложениях А, Б и В вы можете познакомиться с использованием Python в искусстве, бизнесе и науке. Вы найдете как минимум одну область для исследований.

В Сети можно отыскать множество блестящих объектов. Только вы сможете сказать, какие из них являются бижутерией, а какие — серебряными пулями. И даже если вас в данный момент не преследуют оборотни, серебряные пули могут пригодиться. На всякий случай.

Наконец, вы узнаете ответы на все эти раздражающие упражнения в конце каждой главы, детали установки Python и его друзей, а также получите несколько вспомогательных материалов для вещей, которые мне всегда нужно подглядывать. Ваш мозг, скорее всего, лучше подготовлен, но так они всегда будут под рукой.

Приложения

A Пи-Арт

> Ну, искусство есть искусство, не так ли?
> С другой стороны, вода есть вода! Восток есть
> восток, а запад есть запад, и если взять клюкву
> и растолочь ее до консистенции яблочного
> соуса, то по вкусу она будет напоминать
> сливы, не то что толченый ревень.
>
> *Граучо Маркс*

Возможно, вы художник или музыкант. Или, может быть, просто хотите попробовать что-то креативное, выходящее за рамки привычной деятельности.

В первых трех приложениях рассказывается о наиболее распространенных областях применения Python. Если вас интересует одна из этих областей, вы можете почерпнуть из этих глав несколько идей или они подтолкнут вас попробовать что-то новое.

2D-графика

Все языки программирования в какой-то степени работают с компьютерной графикой. Многие мощные платформы в этом приложении для быстродействия были написаны на C или C++, но для продуктивности добавлены библиотеки Python. Начнем с рассмотрения некоторых библиотек для работы с двухмерными изображениями.

Стандартная библиотека

В стандартной библиотеке содержатся всего несколько модулей, связанных с графикой. Рассмотрим два из них:

- `Imghdr`. Этот модуль определяет тип некоторых файлов изображений;
- `Colorsys`. Этот модуль преобразует цвета между разными системами: RGB, YIQ, HSV и HLS.

Если вы загрузили логотип издательства O'Reilly и сохранили его под именем oreilly.png, можете запустить этот код:

```
>>> import imghdr
>>> imghdr.what('oreilly.png')
'png'
```

Чтобы сделать с графикой в Python что-то серьезное, нужно загрузить сторонние пакеты. Давайте их рассмотрим.

PIL и Pillow

Многие годы Python Image Library (PIL, библиотека изображений Python) (http://bit.ly/py-image), несмотря на то что ее нет в стандартной библиотеке, является самой известной библиотекой для обработки двухмерных изображений. Она предшествовала установщикам вроде pip, поэтому был создан «дружественный форк» с названием Pillow (http://pillow.readthedocs.org/). Код для работы с изображениями Pillow совместим с кодом PIL, а его документация хороша, поэтому используем его здесь.

Установить его просто — достаточно ввести следующую команду:

```
$ pip install Pillow
```

Если вы уже устанавливали пакеты libjpeg, libfreetype и zlib, они будут обнаружены и использованы Pillow. На странице с инструкциями по установке (http://bit.ly/pillow-install) вы узнаете больше.

Откроем файл изображения:

```
>>> from PIL import Image
>>> img = Image.open('oreilly.png')
>>> img.format
'PNG'
>>> img.size
(154, 141)
>>> img.mode
'RGB'
```

Несмотря на то что пакет называется Pillow, вы импортируете его как PIL, чтобы код был совместим со старым PIL.

Для того чтобы отобразить изображение на экране с помощью метода show() объекта Image, вы сначала должны установить пакет ImageMagick, описанный в следующем разделе, а затем попробовать вот что:

```
>>> img.show()
```

Изображение, показанное на рис. A.1, открывается в другом окне. (Этот снимок экрана был сделан на компьютере Mac, где функция show() используется

для приложения предварительного просмотра изображений. Отображение ваших окон может быть иным.)

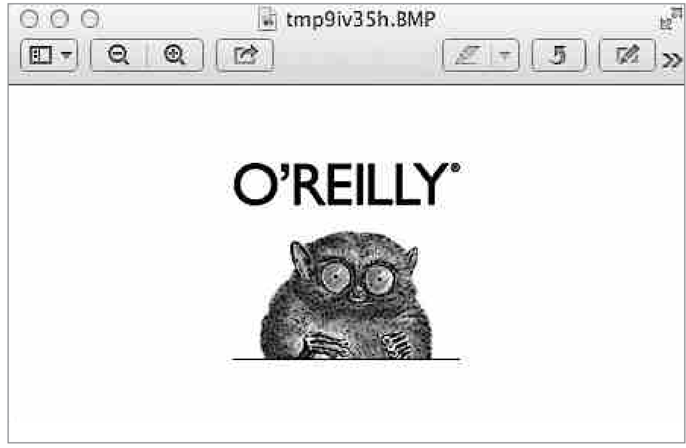


Рис. А.1. Изображение, открытое с помощью библиотеки Python

Обрежем изображение в памяти, сохраним результат как новый объект с именем `img2` и отобразим его.

Изображения всегда измеряются в горизонтальных (x) и вертикальных (y) значениях, один из углов изображения называется стартовой точкой, его значения x и y равны 0. В этой библиотеке `origin(0, 0)` находится в левом верхнем углу изображения, x увеличивается при смещении вправо, а y увеличивается при смещении вниз. Мы хотим задать значения левого края x (`55`), верхнего края y (`70`), правого края x (`85`) и нижнего края y (`100`) для метода `crop()`, поэтому передаем кортеж, содержащий эти значения в соответствующем порядке:

```
>>> crop = (55, 70, 85, 100)
>>> img2 = img.crop(crop)
>>> img2.show()
```

Результат показан на рис. А.2.

Сохраним изображение с помощью метода `save()`. Он принимает имя файла и опциональный путь. Если имя файла имеет суффикс, библиотека использует его, чтобы определить тип. Но вы также можете указать тип файла явно. Для того чтобы сохранить изображение с расширением GIF, сделайте следующее:

```
>>> img2.save('cropped.gif', 'GIF')
>>> img3 = Image.open('cropped.gif')
>>> img3.format
'GIF'
>>> img3.size
(30, 30)
```

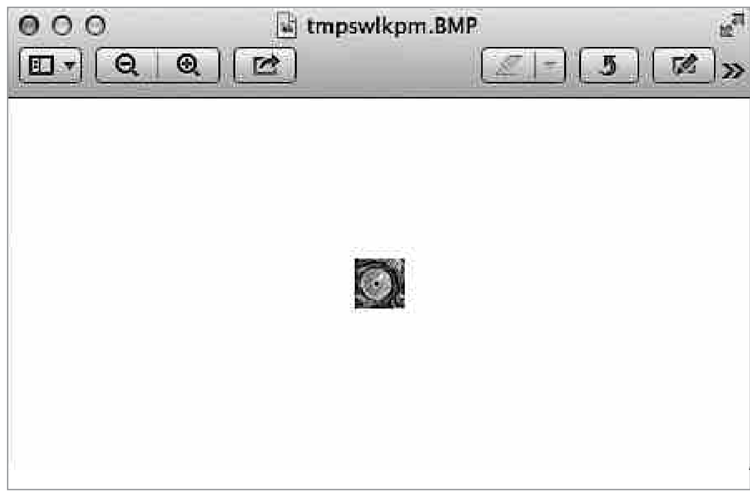


Рис. А.2. Обрезанное изображение

«Улучшим» наш маленький талисман. Сначала загрузим изображение усов (http://bit.ly/moustaches-png) и сохраним его в файл `moustaches.png`. Мы загрузим его, обрежем соответствующим образом, а затем наложим на нашу зверушку:

```
>>> mustache = Image.open('moustaches.png')
>>> handlebar = mustache.crop((316, 282, 394, 310))
>>> handlebar.size
(78, 28)
>>> img.paste(handlebar, (45, 90) )
>>> img.show()
```

На рис. А.3 показан подходящий результат.

Рис. А.3. Наш новый опрятный талисман

Было бы здорово, если бы фон у картинки с усами был прозрачным. О, вот и упражнение для вас! Если вы хотите этим заняться, поищите информацию о прозрачности (transparency) и альфа-канале (alpha channel) в руководстве в `Pillow` (http://bit.ly/pil-fork).

ImageMagick

ImageMagick (http://www.imagemagick.org/) — это комплект программ для конвертирования, изменения и отображения двухмерных изображений. Он существует более 20 лет. Различные библиотеки Python подключены к библиотеке ImageMagick, написанной на C. Самая недавняя из них, поддерживающая Python 3, называется wand (http://docs.wand-py.org/). Для того чтобы установить ее, введите следующую команду:

```
$ pip install Wand
```

С помощью wand вы можете делать примерно то же, что и с помощью `Pillow`:

```
>>> from wand.image import Image
>>> from wand.display import display
>>>
>>> img = Image(filename='oreilly.png')
>>> img.size
(154, 141)
>>> img.format
'PNG'
```

Как и в случае с `Pillow`, эта строка отобразит изображение на экране:

```
>>> display(img)
```

wand позволяет вам поворачивать изображение, изменять его размер, писать текст, рисовать линии и многое другое, что вы можете найти и в `Pillow`. Оба этих пакета имеют хорошие API и документацию.

Графические пользовательские интерфейсы (Graphical User Interface, GUI)

Название содержит слово «графический», но GUI концентрируется скорее на пользовательском интерфейсе: виджетах для представления данных, методах ввода, меню, кнопках и окнах.

Страница «Википедии» *GUI programming* (http://bit.ly/gui-program) и список часто задаваемых вопросов (http://bit.ly/gui-faq) содержат множество примеров GUI, созданных с помощью Python. Начнем с единственного встроенного в стандартную

библиотеку примера — Tkinter (https://wiki.python.org/moin/TkInter). Он прост, но работает на всех платформах и создает естественно выглядящие окна и виджеты.

Рассмотрим небольшую программу, где используется Tkinter, она отображает наш любимый талисман в отдельном окне:

```
>>> import tkinter
>>> from PIL import Image, ImageTk
>>>
>>> main = tkinter.Tk()
>>> img = Image.open('oreilly.png')
>>> tkimg = ImageTk.PhotoImage(img)
>>> tkinter.Label(main, image=tkimg).pack()
>>> main.mainloop()
```

Обратите внимание: мы использовали некоторые модули PIL/Pillow. Вы снова должны увидеть логотип издательства O'Reilly, как показано на рис. А.4.

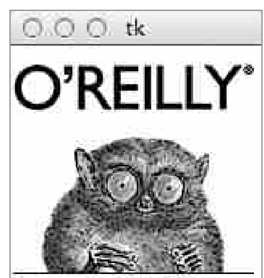


Рис. А.4. Изображение, показанное с помощью библиотеки Tkinter

Для того чтобы окно пропало, нажмите кнопку **Закрыть** или выйдите из интерпретатора Python.

О библиотеке Tkinker вы можете прочитать в tkinter wiki (http://tkinter.unpythonic.net/wiki/) и Python wiki (https://wiki.python.org/moin/TkInter). Теперь мы поговорим о GUI, которые не входят в стандартную библиотеку.

- Qt (http://qt-project.org/). Это профессиональный инструментарий для создания GUI и приложений, созданный около 20 лет назад компанией Trolltech из Норвегии. Он использовался для помощи в создании таких приложений, как Google Earth, Maya и Skype. Он применен также как основа для KDE, графической оболочки Linux. Для Qt существуют две основные библиотеки, работающие с Python: PySide (http://qt-project.org/wiki/PySide) бесплатна (по лицензии LGPL), а PyQt (http://bit.ly/pyqt-info) лицензирована либо с GPL, либо коммерчески. Пользователи Qt видят разницу. Вы можете загрузить PySide с сайтов PyPI (https://pypi.python.org/pypi/PySide) или Qt (http://qt-project.org/wiki/Get-PySide), а также прочесть руководство (http://qt-project.org/wiki/PySide_Tutorials). Загрузить Qt бесплатно можно здесь: http://bit.ly/qt-dl.

- GTK+ (http://www.gtk.org/). Является соперником Qt, он также был использован для создания множества приложений (http://gtk-apps.org/) вроде GIMP и оболочки Gnome для Linux. Для Python используется PyGTK (http://www.pygtk.org/). Чтобы загрузить код, перейдите на сайт PyGTK (http://bit.ly/pygtk-dl), где вы также можете прочитать документацию (http://bit.ly/py-gtk-docs).
- WxPython (http://www.wxpython.org/). Это привязка Python к WxWidgets (http://www.wxwidgets.org/), представляющему еще один крупный пакет, который можно бесплатно загрузить онлайн (http://wxpython.org/download.php).
- Kivy (http://kivy.org/). Это бесплатная современная библиотека для создания мультимедийных интерфейсов пользователя, которые можно переносить на другие платформы — стационарные (Windows, OS X, Linux) и мобильные (Android, iOS). Она имеет поддержку мультитача. Вы можете загрузить ее для всех платформ с сайта Kivy (http://kivy.org/#download). Kivy содержит руководства по разработке приложений (http://bit.ly/kivy-intro).
- The Web. Фреймворки вроде Qt используют встроенные компоненты, но некоторые другие используют Web. Web — это универсальный GUI, который содержит графику (SVG), текст (HTML) и даже мультимедиа (в HTML5). Некоторые инструменты GUI, основанные на нем, содержат RCTK (Remote Control Toolkit) (https://code.google.com/p/rctk/) и Muntjac (http://www.muntiacus.org/). Вы можете создать веб-приложения с любой комбинацией фронтенда (клиентской части) и бэкенда (машинного интерфейса) инструментов. Тонкий клиент позволяет бэкенду делать всю работу. Если доминирует фронтенд, клиент называется толстым или насыщенным, последний эпитет звучит более льстиво. Части приложения могут общаться друг с другом с помощью RESTful API, AJAX и JSON.

Трехмерная графика и анимация

Посмотрите длинные финальные титры любого современного фильма, и вы увидите огромное количество людей, занимавшихся спецэффектами и анимацией. Большинство крупных студий: Walt Disney Animation, ILM, Weta, Dreamworks, Pixar — нанимают людей, имеющих опыт работы с Python. Поищите в Интернете «python анимация работа» или посетите сайт vfxjobs (http://vfxjobs.com/search/) и поищите там «python», чтобы увидеть действующие предложения.

Если вы хотите поэкспериментировать с Python и трехмерной анимацией, мультимедиа и играми, вам следует попробовать Panda3D (http://www.panda3d.org/). Этот движок имеет открытый исходный код и бесплатен даже для коммерческих приложений. Вы можете загрузить версию для своего компьютера с сайта Panda3D (http://bit.ly/dl-panda). Чтобы запустить примеры, измените каталог на `/Developer/Examples/Panda3D`. Каждый подкаталог содержит один или несколько файлов с рас-

ширением .py. Запустите один из них с помощью команды ppython, которая поставляется с Panda3D, например:

```
$ cd /Developer/Examples/Panda3D
$ cd Ball-in-Maze/
$ ppython Tut-Ball-in-Maze.py
DirectStart: Starting the game.
Known pipe types:
  osxGraphicsPipe
(all display modules loaded.)
```

Откроется окно, похожее на то, что показано на рис. А.5.

Рис. А.5. Изображение, показанное с помощью библиотеки Panda3D

Воспользуйтесь мышью, чтобы потрясти коробку и подвигать шарик в лабиринте.

Если все сработало и базовая установка Panda3D выглядит хорошо, можете начать экспериментировать с библиотекой Python.

Рассмотрим простой пример приложения из документации Panda3D (сохраните его как panda1.py):

```
from direct.showbase.ShowBase import ShowBase
class MyApp(ShowBase):
```

```
    def __init__(self):
        ShowBase.__init__(self)
        # Загрузка модели окружения.
        self.environ = self.loader.loadModel("models/environment")
        # Переподчинить модель для отрисовки.
        self.environ.reparentTo(self.render)
        # Применить к модели преобразования масштаба и позиции.
        self.environ.setScale(0.25, 0.25, 0.25)
        self.environ.setPos(-8, 42, 0)
app = MyApp()
app.run()
```

Запустите приложение с помощью следующей команды:

```
$ ppython panda1.py
Known pipe types:
  osxGraphicsPipe
(all display modules loaded.)
```

Откроется окно, содержащее сцену, которая показана на рис. А.6.

Рис. А.6. Масштабированное изображение, показанное с помощью библиотеки Panda3D

Камень и дерево парят над землей. Нажмите кнопку Next (Далее), чтобы продолжить исследовать руководство и исправить эти проблемы.

Далее показаны некоторые пакеты Python для работы с 3D.

- Blender (http://www.blender.org/). Это бесплатное средство создания 3D-анимации и игр. Если вы загрузите и установите его с сайта www.blender.org/download, на ваш компьютер будет установлена копия также Python 3.
- Maya (http://www.autodesk.com/products/autodesk-maya/overview). Это коммерческая система для создания 3D-анимации и графики. Она поставляется вместе с версией Python — в данный момент Python 2.6. Чед Вернон (Chad Vernon) написал о ней бесплатно загружаемую книгу *Python Scripting for Maya Artists* (http://bit.ly/py-maya). Если вы поищете в Интернете информацию о Python и Maya, то сможете найти множество других ресурсов, как бесплатных, так и коммерческих, включая видеоролики.
- Houdini (https://www.sidefx.com/). Это коммерческий пакет, однако вы можете загрузить бесплатную версию, которая называется Apprentice. Как и другие пакеты для анимации, он поставляется с привязкой к Python (http://bit.ly/py-bind).

Диаграммы, графики и визуализация

Python является отличным инструментом для создания диаграмм, графиков и визуализации данных и особенно популярен в научной среде (см. приложение В). Официальный сайт Python содержит обзор таких пакетов (https://wiki.python.org/moin/NumericAndScientific/Plotting). Позвольте мне рассказать чуть более подробно о некоторых из них.

matplotlib

Бесплатная библиотека для создания двухмерных диаграмм matplotlib (http://matplotlib.org/) может быть установлена с помощью следующей команды:

```
$ pip install matplotlib
```

Примеры из галереи (http://matplotlib.org/gallery.html) показывают широту библиотеки matplotlib. Попробуем написать такое же приложение для показа изображений (результаты можно увидеть на рис. А.7) только для того, чтобы увидеть, как будут выглядеть код и презентация:

```
import matplotlib.pyplot as plot
import matplotlib.image as image
img = image.imread('oreilly.png')
plot.imshow(img)
plot.show()
```

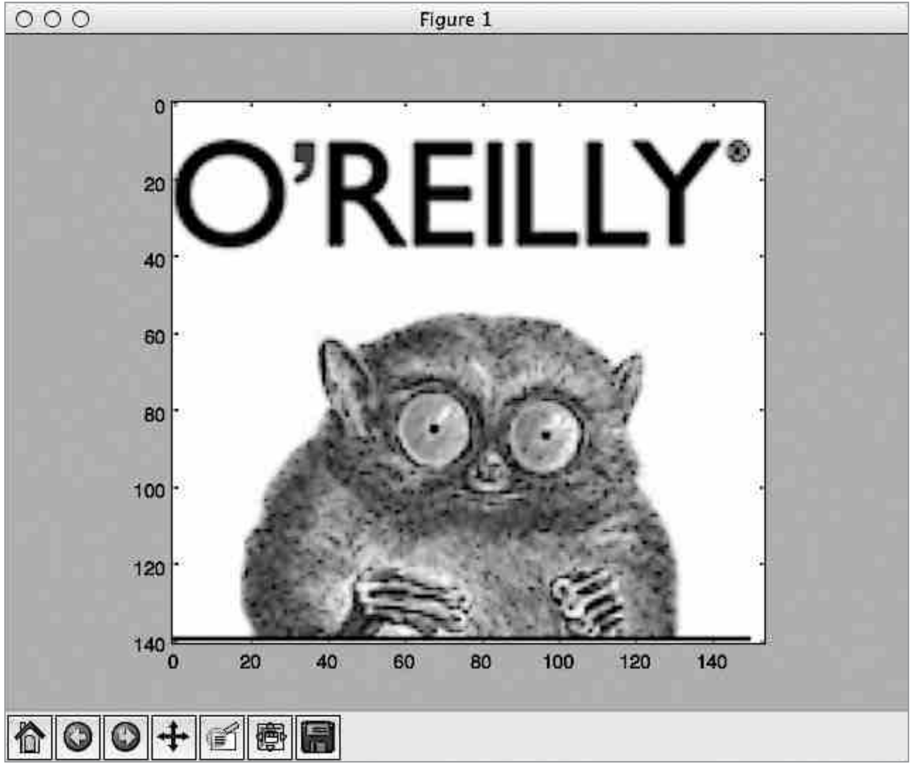


Рис. А.7. Изображение, показанное с помощью библиотеки matplotlib

В приложении В мы еще вернемся к matplotlib — она тесно связана с NumPy и другими научными приложениями.

bokeh

В ранние дни существования Интернета разработчики генерировали графику на сервере и давали браузеру URL для доступа к ней. В недавнее время JavaScript повысил свою производительность и получил инструменты генерации графики на стороне клиента, такие как D3. Пару страниц назад я говорил о возможности использовать Python как часть архитектуры фронтенд-бэкенд-графики и GUIs. Новый инструмент, который называется bokeh (http://bokeh.pydata.org/), совмещает плюсы Python (крупные наборы данных, простота использования) и JavaScript (интерактивность, меньшая латентность графики). Он делает акцент на быстрой визуализации крупных наборов данных.

Если вы уже установили необходимые для bokeh пакеты (NumPy, Pandas и Redis), можете установить и его самого, введя следующую команду:

```
$ pip install bokeh
```

(NumPy и Pandas в действии вы сможете увидеть в приложении В.)

Или же можете установить все сразу с сайта Bokeh (http://bit.ly/bokeh-dl). Несмотря на то что на сервере запущен matplotlib, bokeh в основном работает в браузере и может пользоваться всеми новыми возможностями клиентской стороны. Нажмите на любое изображение в галерее (http://bokeh.pydata.org/docs/gallery.html), чтобы получить интерактивное представление дисплея и его код.

Игры

Python хорошо приспособлен для выпаса данных, и вы уже видели в этом приложении, что он хорош для работы с мультимедиа. Но как насчет игр?

Оказывается, Python — это настолько хорошая платформа для написания игр, что об этом пишут книги. Вот несколько из них:

- *Invent Your Own Computer Games with Python*, автор Эл Свайгарт (Al Sweigart) (http://inventwithpython.com/);
- *The Python Game Book*, автор Хорст Йенс (Horst Jens) (книга в формате docuwiki) (http://thepythongamebook.com/).

В Python wiki вы можете найти статью (https://wiki.python.org/moin/PythonGames), в которой содержится еще большее количество ссылок.

Самой известной платформой для написания игр, скорее всего, является pygame (http://pygame.org/). Вы можете загрузить исполняемый установщик для своей платформы с сайта Pygame и прочесть построчный пример создания игры (http://bit.ly/line-chimp).

Аудио и музыка

Что насчет звука, музыки и котов, которые поют *Jingle Bells*?

Сейчас мы поговорим о первых двух пунктах.

Стандартная библиотека содержит несколько рудиментарных модулей для работы с аудио в разделе служб мультимедиа (http://docs.python.org/3/library/mm.html). Кроме того, существует страница, на которой обсуждаются сторонние модули (https://wiki.python.org/moin/Audio).

Следующие библиотеки могут помочь вам генерировать музыку:

- pyknon (https://github.com/kroger/pyknon) — используется в книге *Music for Geeks and Nerds*, автор Педро Крогер (Pedro Kroger) (Create-Space);
- mingus (https://code.google.com/p/mingus/) — это музыкальный секвенсер, который может читать и создавать MIDI-файлы;
- remix (http://echonest.github.io/remix/python.html) оправдывает свое название — это API для создания ремиксов. Одним из примеров его использования является `morecowbell.dj`, который добавляет в загруженные песни звук колокольчиков;
- sebastian (https://github.com/jtauber/sebastian/) — это библиотека для музыкальной теории и анализа;

- Piano (http://bit.ly/py-piano) — позволяет вам играть на пианино с помощью клавиатуры, а именно клавиш C, D, E, F, A, B и C.

Наконец, рассмотрим библиотеки, которые помогут вам организовать свою коллекцию или дать вам доступ к музыкальным данным:

- Beets (http://beets.radbox.org/) — управляет вашей коллекцией музыки;
- API Echonest (http://developer.echonest.com/) — позволяет получить доступ к метаданным музыки;
- Monstermash (http://bit.ly/mm-karlgrz) — объединяет фрагменты песен, он создан с помощью Echonest, Flask, ZeroMQ и Amazon EC2;
- Shiva (http://bit.ly/shiva-api) — это RESTful API и сервер (https://github.com/tooxie/shiva-server), предназначенные для организации коллекции по вашему усмотрению;
- получите обложки альбомов для своей музыки по адресу http://jameh.github.io/mpd-album-art/.

Б За работой

— Дела! — вскричал призрак, снова заламывая руки. — Забота о ближнем — вот что должно было стать моим делом...

Чарльз Диккенс. Рождественские повести

Униформа бизнесмена — костюм и галстук. Но по какой-то причине, когда он решает взяться за дело, он вешает пиджак на спинку стула, ослабляет галстук, закатывает рукава и наливает себе кофе. В то же время бизнесвумен без особого парада просто выполняет свою работу. Может быть, возьмет латте.

В области бизнеса мы используем все технологии из предыдущих глав — базы данных, веб-системы и сети. Продуктивность Python делает его более популярным и у корпораций (http://bit.ly/py-enterprise), и у стартапов (http://bit.ly/py-startups).

Бизнес долго искал серебряные пули, чтобы убивать давно преследовавших его оборотней: несовместимые форматы файлов, скрытые сетевые протоколы, обязательные языки и общую нехватку точной документации. Однако сегодня мы имеем доступ к технологиям и приемам, которые могут масштабироваться и взаимодействовать друг с другом. Бизнес может создавать более быстрые, более дешевые эластичные приложения, пользуясь:

- динамическими языками вроде Python;
- Сетью как универсальным графическим интерфейсом пользователя;
- RESTful API как независимыми от языка интерфейсами служб;
- реляционными и NoSQL-базами данных;
- «большими данными» и аналитикой;
- облаками для развертывания и экономии капитала.

The Microsoft Office Suite

Бизнес сильно зависит от приложений Microsoft Office и форматов файлов. Несмотря на то что эти библиотеки не очень широко известны и в некоторых случаях

плохо задокументированы, они могут пригодиться. Рассмотрим некоторые из них, они работают с документами Microsoft Office:

- docx (https://pypi.python.org/pypi/docx). Эта библиотека создает, считывает и записывает файлы для Microsoft Office Word 2007 с расширением `.docx`;
- python-excel (http://www.python-excel.org/). Здесь с помощью PDF-руководства рассматриваются модули `xlrd`, `xlwt` и `xlutils`. Excel также может читать и записывать файлы, содержащие значения, разделенные запятыми (Comma-Separated Value, CSV), с которыми вы уже умеете работать с помощью стандартного модуля `csv`;
- oletools (http://bit.ly/oletools). Эта библиотека извлекает данные из форматов Office.

Следующие модули автоматизируют приложения операционной системы Windows:

- pywin32 (http://sourceforge.net/projects/pywin32/). Этот модуль автоматизирует множество приложений Windows. Однако он использует только Python 2 и имеет скудную документацию — обратите внимание на эти статьи: http://bit.ly/pywin32-lib и http://bit.ly/pywin-mo;
- pywinauto (https://code.google.com/p/pywinauto/). Этот модуль также автоматизирует приложения Windows и использует только Python 2 — обратите внимание на эту статью;
- swapy (https://code.google.com/p/swapy/). Генерирует код на Python для pywinauto из встроенных элементов управления.

OpenOffice (http://openoffice.org/) — это альтернатива Office, имеющая открытый исходный код. Она работает в операционных системах Linux, Unix, Windows и OS X и также читает и записывает форматы файлов Office. Кроме того, это приложение устанавливает версию Python 3 для себя. Вы можете запрограммировать OpenOffice с помощью Python (https://wiki.openoffice.org/wiki/Python) и библиотеки PyUNO (http://www.openoffice.org/udk/python/python-bridge.html).

OpenOffice принадлежал компании Sun Microsystems, и когда компания Oracle приобрела компанию Sun, некоторые люди стали опасаться, что OpenOffice в будущем станет недоступен. В результате появился LibreOffice (https://www.libreoffice.org/). В DocumentHacker (http://bit.ly/docu-hacker) вы можете прочитать об использовании библиотеки Python UNO вместе с LibreOffice.

Для создания OpenOffice и LibreOffice пришлось выполнить реверс-инжиниринг форматов файлов Microsoft, что не так легко сделать. Модуль Universal Office Converter (http://dag.wiee.rs/home-made/unoconv/) зависит от библиотеки UNO в OpenOffice или LibreOffice. Он может преобразовывать файлы многих форматов: документы, электронные таблицы, графику и презентации.

Если у вас имеется таинственный файл, `python-magic` (https://github.com/ahupp/python-magic) может угадать его формат, проанализировав определенные последовательности байтов.

Библиотека python open document (http://appyframework.org/pod.html) позволяет вам предоставить код Python внутри шаблонов для создания динамических документов.

Формат PDF распространен также в области бизнеса, несмотря на то что он создан не компанией Microsoft. Движок ReportLab (http://www.reportlab.com/opensource/) имеет бесплатную и коммерческую версии генератора PDF, созданные с помощью Python. Если вам нужно отредактировать PDF-файл, вы можете найти помощь на сайте StackOverflow (http://bit.ly/add-text-pdf).

Выполняем бизнес-задачи

Вы можете найти модуль Python практически для чего угодно. Посетите сайт PyPI (https://pypi.python.org/pypi) и введите что-нибудь в строку поиска. Многие модули являются интерфейсами для общедоступных API различных служб. Вам могут быть интересны примеры, связанные с бизнес-задачами:

- отправка через Fedex (https://github.com/gtaylor/python-fedex) или UPS (https://github.com/openlabs/PyUPS);
- отправка почты с помощью API stamps.com (https://github.com/jzempel/stamps);
- прочтите дискуссию о применении Python для корпоративного интеллекта (http://bit.ly/py-biz);
- если кофемашины слетают с полок в Аноке, это результат действий покупателя или полтергейст? Cubes — это веб-сервер и браузер данных Online Analytical Processing (OLAP) (http://cubes.databrewery.org/);
- OpenERP — это крупная коммерческая система Enterprise Resource Planning (ERP) (https://www.openerp.com/), написанная с помощью Python и JavaScript, содержащая тысячи модулей-надстроек.

Обработка бизнес-данных

Бизнес очень любит данные. К сожалению, во многих отраслях бизнеса с данными принято обращаться очень своеобразно, словно чтобы специально усложнить вам работу.

Электронные таблицы стали хорошим изобретением, и с течением времени бизнес пристрастился к ним. Многие непрограммисты стали заниматься программированием, поскольку эти таблицы стали называть макросами, а не программами. Но Вселенная постоянно расширяется, и данные пытаются выдержать ее темп. Более старые версии Excel были ограничены 65 536 рядами, а более новые не могли обработать больше миллиона. Когда данные организаций переросли один компьютер, ситуация стала похожа на то, как если бы штат перерос сотню человек, — внезапно вам становятся нужны новые слои, промежуточные уровни и коммуникация.

Программы, работающие с большим количеством данных, появились не из-за того, что данные на одном компьютере значительно разрослись, — они стали результатом попытки обобщить все данные, появляющиеся в бизнесе. Реляционные базы данных обрабатывают миллионы рядов, не взрываясь при этом, но они могут работать только с определенным количеством рядов или обновлений за раз. Старые добрые текстовые или бинарные файлы могут занимать гигабайты памяти, но если вам нужно обработать их все одновременно, требуется иметь достаточное количество памяти. Традиционное программное обеспечение для этого не подходит. Компаниям вроде Google и Amazon пришлось изобрести решения, позволяющие работать с данными такого масштаба. Netflix — это пример такого решения, созданный в облаке AWS от Amazon, который использует Python для объединения RESTful API, безопасности, развертывания и баз данных.

Извлечение, преобразование и загрузка

Подводные части айсбергов данных содержат всю работу по получению данных. Если говорить языком бизнеса, распространенными терминами являются «извлечение», «преобразование», «загрузка» (extract, transform, load, ETL). Синонимы вроде «выпас данных» могут создать впечатление, будто вы приручаете непокорного зверя — такая метафора может оказаться близкой. Эта задача может показаться решенной с точки зрения инженерии, но по большей части это искусство. Мы рассмотрим науку о данных более подробно в приложении В, поскольку именно на эту область разработчики тратят бо́льшую часть времени.

Если вы видели фильм «Волшебник страны Оз», то, помимо летающих обезьян, должны помнить фрагмент в конце, когда добрая волшебница сказала Дороти, что она может отправиться домой в Канзас, просто стукнув каблучком о каблучок. Даже когда я был моложе, я думал: «И она говорит об этом только сейчас?!» Хотя теперь я понимаю, что фильм был бы гораздо короче, если бы волшебница дала такой совет раньше.

Вот только мы живем не в фильме, здесь мы говорим о мире бизнеса, где сокращение времени на выполнение задач — это хорошо. Позвольте мне поделиться с вами парой советов. Большинство инструментов, которые вам понадобятся для повседневной работы с данными в бизнесе, вы уже знаете, поскольку прочитали о них в этой книге. Среди таких инструментов высокоуровневые структуры данных вроде словарей и объектов, тысячи стандартных и сторонних библиотек, а также сообщество экспертов, которое можно найти в поисковике.

Если вы программист, который пишет программы для бизнеса, ваш рабочий поток практически всегда содержит следующее.

1. Извлечение данных из файлов странных форматов или баз данных.
2. «Очистку» данных, которая охватывает множество областей, заполненных острыми объектами.
3. Преобразование дат, времени и наборов символов.
4. Выполнение каких-либо действий над данными.

5. Сохранение полученного результата в базе данных.
6. Откат к первому шагу: намылить, смыть, повторить.

Рассмотрим пример: вам нужно переместить данные из электронной таблицы в базу данных. Вы можете сохранить таблицу в формате CSV и использовать библиотеки Python, показанные в главе 8. Или же найти модуль, который считывает непосредственно бинарный формат электронной таблицы. Ваши пальцы знают, как набрать строку `python excel` в поисковике и найти сайты вроде Working with Excel files in Python (http://www.python-excel.org/). Вы можете установить один из необходимых пакетов с помощью pip и найти драйвер базы данных для Python, чтобы выполнить последнюю часть задания. Я упоминал SQLAlchemy и непосредственные драйверы базы данных в той же главе. Теперь вам нужно написать промежуточный код, и именно здесь структуры данных и библиотеки Python могут сэкономить ваше время.

Попробуем выполнить поставленную задачу, а затем выполнить ее снова, но уже используя библиотеки, которые позволяют сэкономить немного времени. Мы считаем CSV-файл, агрегируем все числа в одной колонке, упорядочив их по уникальному значению, и выведем результат на экран. Если бы мы решали задачу с помощью SQL, то использовали бы `SELECT`, `JOIN` и `GROUP BY`.

Для начала рассмотрим файл `zoo.csv`, который содержит следующую информацию: тип животного, сколько раз оно укусило посетителя, количество потребовавшихся швов и сумма, которую мы заплатили посетителю за то, чтобы он ничего не говорил журналистам:

```
animal,bites,stitches,hush
bear,1,35,300
marmoset,1,2,250
bear,2,42,500
elk,1,30,100
weasel,4,7,50
duck,2,0,10
```

Мы хотим узнать, какое животное обходится нам дороже всего, поэтому агрегируем общую сумму взяток для каждого типа животного (количеством укусов и швов пусть займется интерн). Мы используем модуль `csv`, который рассматривали в разделе «CSV» главы 8 и счетчик `Counter` из подраздела «Подсчитываем элементы с помощью функции Counter()» раздела «Стандартная библиотека Python» главы 5. Сохраните этот код как `zoo_counts.py`:

```
import csv
from collections import Counter
counts = Counter()
with open('zoo.csv', 'rt') as fin:
    cin = csv.reader(fin)
    for num, row in enumerate(cin):
        if num > 0:
            counts[row[0]] += int(row[-1])
for animal, hush in counts.items():
    print("%10s %10s" % (animal, hush))
```

Мы пропустили первый ряд, поскольку в нем содержались только имена колонок. `counts` — это объект типа `Counter`, он управляет инициализацией суммы для каждого типа животного, устанавливая ее равной `0`. Мы также применили форматирование, чтобы выровнять выводимую информацию по правому краю. Запустим наш код:

```
$ python zoo_counts.py
     duck         10
      elk        100
     bear        800
   weasel         50
 marmoset        250
```

Ага! Это был медведь! Он был нашим основным подозреваемым все время, но теперь у нас есть доказательства.

Далее воссоздадим этот пример с инструментальным средством для обработки данных `Bubbles` (http://bubbles.databrewery.org/). Вы можете установить его, введя следующую команду:

```
$ pip install bubbles
```

Он требует наличия SQLAlchemy. Если у вас его нет, вам поможет команда `pip install sqlalchemy`. Так выглядит тестовая программа (назовите файл `bubbles1.py`), адаптированная из документации (http://bit.ly/py-bubbles):

```
import bubbles
p = bubbles.Pipeline()
p.source(bubbles.data_object('csv_source', 'zoo.csv', infer_fields=True))
p.aggregate('animal', 'hush')
p.pretty_print()
```

А теперь момент истины:

```
$ python bubbles1.py
 2014-03-11 19:46:36,806 DEBUG calling aggregate(rows)
 2014-03-11 19:46:36,807 INFO called aggregate(rows)
 2014-03-11 19:46:36,807 DEBUG calling pretty_print(records)
 +--------+--------+------------+
 |animal  |hush_sum|record_count|
 +--------+--------+------------+
 |duck    |      10|           1|
 |weasel  |      50|           1|
 |bear    |     800|           2|
 |elk     |     100|           1|
 |marmoset|     250|           1|
 +--------+--------+------------+
 2014-03-11 19:46:36,807 INFO called pretty_print(records)
```

Если вы прочитали документацию, то можете избежать вывода на экран строк с отладочной информацией и, возможно, изменить формат таблицы.

Сравнивая два примера, можно заметить, что пример с `bubbles` использовал один вызов функции (`aggregate`), чтобы заменить чтение и подсчет данных в формате CSV вручную. В зависимости от того, что вам нужно, инструментальные средства работы с данными могут сберечь вам много времени.

В более реалистичном примере наш файл может содержать тысячи строк (он становится опасным), в которых можно встретить опечатки вроде `bare`, запятые в числах и т. д. Чтобы найти хорошие примеры практических задач, связанных с данными, и их решений на Python и Java, обратитесь к книге Грега Уилсона (Greg Wilson) *Data Crunching: Solve Everyday Problems Using Java, Python, and More* (издательство Pragmatic Bookshelf).

Инструменты очистки данных могут сэкономить кучу времени, и Python имеет множество таких инструментов. Например, PETL (http://petl.readthedocs.org/) позволяет выполнять извлечение и переименование рядов и колонок. В приложении В рассматриваются особенно полезные инструменты для работы с данными: Pandas, NumPy и IPython. В дополнение к их широкой известности в научной среде они стали популярными инструментами среди разработчиков, работающих с финансами и данными. На конференции PyData в 2012 году компания AppData (http://bit.ly/py-big-data) рассматривала, как эти три и другие инструменты Python помогают обработать 15 Тбайт данных ежедневно. Это не опечатка — Python может обрабатывать очень большие объемы реальных данных.

Дополнительные источники информации

Иногда вам нужны данные, которые появляются где-то в другом месте. Рассмотрим некоторые источники данных из области бизнеса и правительственной информации.

- data.gov (https://www.data.gov/). Открывает доступ к тысячам наборов данных и инструментов. Его API созданы на основе CKAN, системы управления данными Python.
- Opening government with Python (http://sunlightfoundation.com/). Посмотрите видеоролики (http://bit.ly/opengov-py) и слайды (http://goo.gl/8Yh3s).
- python-sunlight (http://bit.ly/py-sun). Библиотеки, позволяющие получить доступ к Sunlight API (http://sunlightfoundation.com/api/).
- Froide (http://stefanw.github.io/froide/). Платформа, основанная на django, для управления свободой информационных запросов.
- 30 places to find open data on the Web (http://blog.visual.ly/data-sources/). Различные полезные ссылки.

Python в области финансов

С недавнего времени в финансовой индустрии развился значительный интерес к Python. Адаптируя программное обеспечение, показанное в приложении В, а также разрабатывая собственное, группа *Pythonquants* создает новое поколение финансовых инструментов.

- Quantitative economics (http://quant-econ.net/). Этот инструмент предназначен для экономического моделирования, с его помощью вы можете выполнить множество расчетов, применяя Python.
- Python for finance (http://www.python-for-finance.com/). Этот сайт представляет книгу Ивса Хилпиша (Yves Hilpisch) *Derivatives Analytics with Python: Data Analytics, Models, Simulation, Calibration, and Hedging* (издательство Wiley).
- Quantopian (https://www.quantopian.com/). Это интерактивный сайт, на котором вы можете писать собственный код Python и запускать его для архива данных об акциях.
- PyAlgoTrade (http://gbeced.github.io/pyalgotrade/). Еще один инструмент для тестирования на исторических данных, но уже для вашего собственного компьютера.
- Quandl (http://www.quandl.com/). Используйте этот инструмент для поиска в миллионах единиц финансовых данных.
- Ultra-finance (https://code.google.com/p/ultra-finance/). Библиотека, содержащая набор информации об акциях, которая обновляется в реальном времени.
- *Python for Finance* (издательство O'Reilly). Книга Ивса Хилпиша (Yves Hilpisch), содержащая примеры финансового моделирования, написанные на Python.

Безопасность бизнес-данных

Безопасность — это особая забота для бизнеса. Целые книги посвящены этой теме, поэтому мы лишь упомянем несколько советов, связанных с Python.

- В разделе «Scapy» главы 11 рассматривается `scapy`, язык на базе Python для экспертизы пакетов. Он используется для объяснения некоторых крупных сетевых атак.
- Сайт Python Security (http://www.pythonsecurity.org/) содержит дискуссии на тему безопасности, детали некоторых модулей Python и вспомогательные материалы.
- Книга Ти Джея О'Коннора (TJ O'Connor) *Violent Python* (с подзаголовком *A Cookbook for Hackers, Forensic Analysts, Penetration Testers and Security Engineers*) (издательство Syngress) — это широкий обзор Python и компьютерной безопасности.

Карты

Карты стали полезными для многих бизнесов. Python очень хорошо рисует карты, поэтому мы потратим немного времени на эту тему. Менеджеры любят графики, и если вы можете быстро нарисовать красивую карту сайта вашей организации, это вам не повредит.

В ранние дни существования Интернета я посещал экспериментальный сайт по созданию карт у Xerox. Когда появились крупные сайты вроде Google Maps, они стали откровением (к тому же вызывали мысль: «Почему я не подумал об этом и не заработал миллионы?»). Теперь службы картографии и службы, основанные на определении местоположения, практически везде, они особенно удобны в мобильных устройствах.

Здесь пересекается множество терминов: картография, GIS (geographic information system — географическая информационная система), GPS (Global Positioning System — глобальная система позиционирования), анализ геопространства и многие другие. Блог Geospatial Python (http://bit.ly/geospatial-py) воплощает образ системы размером с «800-фунтовую гориллу» GDAL/OGR, GEOS и PROJ.4 (проекции) и вспомогательные системы, представленные как обезьяны.

Многие из этих служб имеют интерфейсы Python. Поговорим о некоторых из них, начиная с самых простых форматов.

Форматы

Мир картографии имеет множество форматов: векторный (линии), растровый (изображения), метаданные (слова) и их комбинации.

Esri, первая географическая система, изобрела формат шейп-файл более 20 лет назад. Файл формата шейп-файл содержит несколько файлов, содержащих как минимум следующую информацию:

- `.shp` — информация о фигуре (вектор);
- `.shx` — индекс формы;
- `.dbf` — база данных атрибутов.

Рассмотрим некоторые модули для работы с такими файлами.

- pyshp (https://code.google.com/p/pyshp/) — это библиотека для работы с шейп-файлами, написанная полностью на Python.
- shapely (http://toblerity.org/shapely/) решает геометрические вопросы наподобие «Какие строения в этом городе через 50 лет окажутся в зоне наводнения?».
- fiona (https://github.com/Toblerity/Fiona) оборачивает библиотеку OGR, которая работает с шейп-файлами и другими векторными форматами.
- kartograph (http://kartograph.org/) отрисовывает шейп-файлы в карты формата SVG на сервере или клиенте.

- basemap (http://matplotlib.org/basemap/) наносит двухмерные данные на карты и использует matplotlib.
- cartopy (http://scitools.org.uk/cartopy/docs/latest/) использует matplotlib и shapely для того, чтобы рисовать карты.

Получим шейп-файл для нашего следующего примера. Посетите страницу http://bit.ly/cultural-vectors. В разделе Admin 1 — States and Provinces нажмите зеленую кнопку download states and provinces (загрузить штаты и провинции), чтобы загрузить архив. После загрузки разархивируйте файл, вы должны увидеть такой результат:

```
ne_110m_admin_1_states_provinces_shp.README.html
ne_110m_admin_1_states_provinces_shp.sbn
ne_110m_admin_1_states_provinces_shp.VERSION.txt
ne_110m_admin_1_states_provinces_shp.sbx
ne_110m_admin_1_states_provinces_shp.dbf
ne_110m_admin_1_states_provinces_shp.shp
ne_110m_admin_1_states_provinces_shp.prj
ne_110m_admin_1_states_provinces_shp.shx
```

Мы будем использовать эти файлы в наших примерах.

Нарисуем карту

Для прочтения шейп-файла вам понадобится эта библиотека:

```
$ pip install pyshp
```

Теперь введите текст программы, `map1.py`, который я модифицировал из статьи в блоге Geospatial Python (http://bit.ly/raster-shape):

```
def display_shapefile(name, iwidth=500, iheight=500):
    import shapefile
    from PIL import Image, ImageDraw
    r = shapefile.Reader(name)
    mleft, mbottom, mright, mtop = r.bbox
    # map units
    mwidth = mright - mleft
    mheight = mtop - mbottom
    # scale map units to image units
    hscale = iwidth/mwidth
    vscale = iheight/mheight
    img = Image.new("RGB", (iwidth, iheight), "white")
    draw = ImageDraw.Draw(img)
    for shape in r.shapes():
        pixels = [
            (int(iwidth - ((mright - x) * hscale)), int((mtop - y) * vscale))
            for x, y in shape.points]
        if shape.shapeType == shapefile.POLYGON:
            draw.polygon(pixels, outline='black')
        elif shape.shapeType == shapefile.POLYLINE:
```

```
            draw.line(pixels, fill='black')
    img.show()
if __name__ == '__main__':
    import sys
    display_shapefile(sys.argv[1], 700, 700)
```

Эта программа считывает шейп-файл и проходит по отдельным фигурам. Я ищу только два типа фигур: многоугольник, который соединяет последнюю точку с начальной, и ломаную линию, которая этого не делает. Я строил свою логику на основе оригинальной статьи и беглого просмотра документации pyshp, поэтому вполне уверен в том, что знаю, как работает программа. Иногда вам просто нужно начать и затем справляться с возникающими проблемами.

Запустим наш код. Аргументом станет базовое имя для шейп-файла, не содержащее расширения:

```
$ python map1.py ne_110m_admin_1_states_provinces_shp
```

Вы должны увидеть что-то похожее на рис. Б.1.

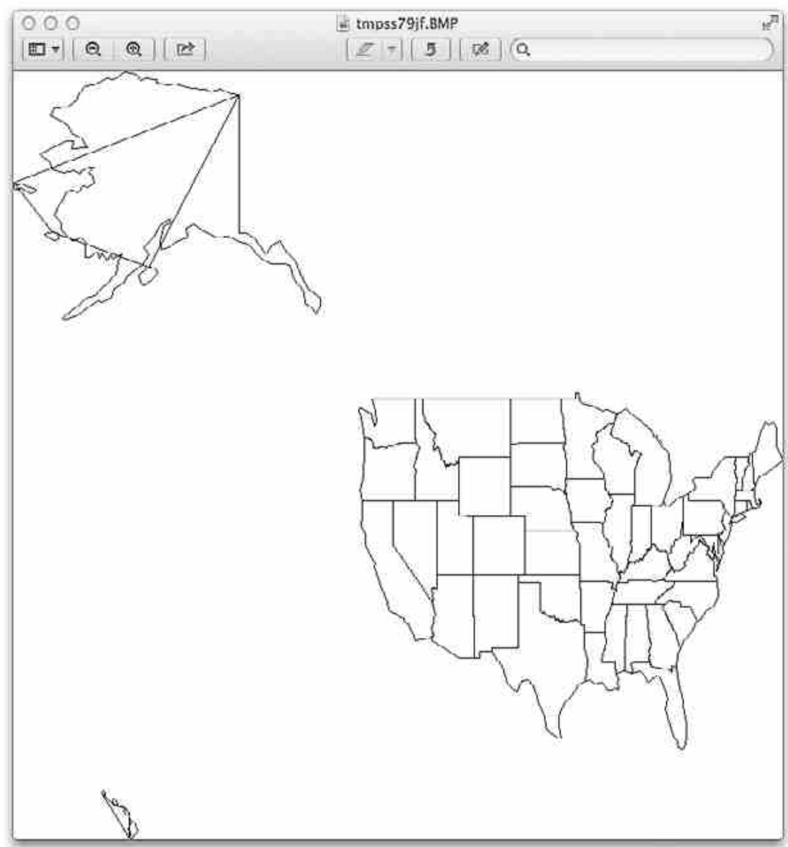


Рис. Б.1. Предварительная карта

Что ж, программа нарисовала карту, которая напоминает Соединенные Штаты, но:

- между Аляской и Гавайями видим что-то вроде растрепанных кошкой ниток — это баг;
- страна сплющена, а мне нужна проекция;
- картинка не очень красивая — мне нужно задать стиль.

По поводу первого пункта: в логике программы есть ошибка, но что мне делать? В главе 12 рассматриваются советы для разработчиков, включая информацию об отладке, но мы можем рассмотреть и другие варианты. Я мог бы написать несколько тестов и работать, пока не исправлю ошибку, или же могу применить какую-нибудь другую библиотеку для создания карты. Возможно, более высокоуровневое решение поможет мне справиться со всеми тремя проблемами (лишние линии, сплющенный вид и примитивный стиль).

Вот несколько ссылок на картографическое программное обеспечение Python.

- basemap (http://matplotlib.org/basemap/). Основана на matplotlib, предназначена для рисования карт и перекрытия их данных.
- mapnik (http://mapnik.org/). Библиотека, написанная на C++, имеющая привязку к Python. Используется для создания векторных (линии) и растровых (изображения) карт.
- tilemill (https://www.mapbox.com/tilemill/). Студия дизайна карт, основанная на mapnik.
- Vincent (http://vincent.readthedocs.org/). Преобразуется в Vega, инструмент визуализации JavaScript, смотрите руководство http://wrobstory.github.io/2013/10/mapping-data-python.html.
- Python for ArcGIS (http://bit.ly/py-arcgis). Ссылки на ресурсы Python для коммерческого продукта ArcGIS фирмы Esri.
- Spatial analysis with python (http://bit.ly/spacial-analysis). Ссылки на руководства, пакеты и видеоролики.
- Using geospatial data with python (http://bit.ly/geos-py). Видеопрезентации.
- So you'd like to make a map using Python (http://bit.ly/pythonmap). Использует pandas, matplotlib, shapely и другие модули Python для создания карт с расположением памятных плит на зданиях.
- *Python Geospatial Development* (Packt). Книга Эрика Вестры (Eric Westra), содержащая примеры использования mapnik и других инструментов.
- *Learning Geospatial Analysis with Python* (Packt). Еще одна книга. Ее написал Джоэл Лохед (Joel Lawhead). Он сделал обзор форматов и библиотек, а также включил геопространственные алгоритмы.

Все эти модули создают красивые карты, но их труднее установить и изучить. Некоторые из них зависят от другого ПО, которого вы еще не видели, вроде numpy

и pandas. Стоит ли овчинка выделки? Как разработчикам, нам часто нужно совершать подобные сделки, основываясь на неполной информации. Если вам интересны карты, попробуйте загрузить и установить один из этих пакетов и посмотреть, что вы можете с его помощью сделать. Или можете избежать установки ПО и попробовать соединиться с API удаленного сервера самостоятельно — в главе 9 показывается, как можно соединяться с веб-серверами и декодировать ответы JSON.

Приложения и данные

Мы говорили о рисовании карт, но с данными о картах вы можете сделать гораздо большее. Геокодирование преобразует адреса в географические координаты и наоборот. Существует множество геокодирующих API (http://www.programmableweb.com/apitag/geocoding) (их сравнение вы можете увидеть на сайте http://bit.ly/free-geo-api) и библиотек Python: geopy (https://code.google.com/p/geopy/), pygeocoder (https://pypi.python.org/pypi/pygeocoder) и googlemaps (http://py-googlemaps.sourceforge.net/). Если вы авторизуетесь с помощью Google или другого источника, чтобы получить ключ для API, вы сможете получить доступ к другим службам, выполняющим, например, пошаговое прокладывание маршрутов путешествий или локальный поиск.

Вот несколько ресурсов, касающихся отображения данных.

- http://www.census.gov/geo/maps-data/. Обзор файлов карт U.S. Census Bureau.
- http://www.census.gov/geo/maps-data/data/tiger.html. Множество географических и демографических карт.
- http://wiki.openstreetmap.org/wiki/Potential_Datasources. Мировые ресурсы.
- http://www.naturalearthdata.com/. Векторные и растровые данные карт в трех масштабах.

Нам следует упомянуть здесь Data Science Toolkit (http://www.datasciencetoolkit.org/). Он содержит бесплатные возможности двухстороннего геокодирования, вычисления координат политических границ и статистики и даже больше. Вы можете загрузить все данные и ПО как виртуальную машину и запустить их отдельно на своем компьютере.

B Py в науке

В последние годы в основном из-за ПО, показанного в этом приложении, Python стал очень популярен среди ученых. Если вы и сами ученый или студент, то, возможно, пользовались инструментами вроде MatLab и R или традиционными языками вроде Java, C или C++. В этом приложении вы увидите, что Python стал отличной платформой для научного анализа и публикации результатов.

Математика и статистика в стандартной библиотеке

Для начала вернемся к стандартной библиотеке и рассмотрим некоторые особенности и модули, которые мы проигнорировали.

Математические функции

Python имеет множество математических функций в стандартной библиотеке math (https://docs.python.org/3/library/math.html). Просто введите `import math`, чтобы получить к ним доступ из своих программ.

Она содержит такие константы, как `pi` и `e`:

```
>>> import math
>>> math.pi
>>> 3.141592653589793
>>> math.e
2.718281828459045
```

В основном код состоит из функций, поэтому рассмотрим наиболее полезные из них.

Функция `fabs()` возвращает абсолютное значение своего аргумента:

```
>>> math.fabs(98.6)
98.6
>>> math.fabs(-271.1)
271.1
```

Получаем округление вниз (`floor()`) и вверх (`ceil()`) некоторого числа:

```
>>> math.floor(98.6)
98
>>> math.floor(-271.1)
-272
>>> math.ceil(98.6)
99
>>> math.ceil(-271.1)
-271
```

Высчитываем факториал (в математике это выглядит как `n!`) с помощью функции `factorial()`:

```
>>> math.factorial(0)
1
>>> math.factorial(1)
1
>>> math.factorial(2)
2
>>> math.factorial(3)
6
>>> math.factorial(10)
3628800
```

Получаем натуральный логарифм аргумента с помощью функции `log()`:

```
>>> math.log(1.0)
0.0
>>> math.log(math.e)
1.0
```

Если вы хотите задать другое основание логарифма, передайте его как второй аргумент:

```
>>> math.log(8, 2)
3.0
```

Функция `pow()` возвращает противоположный результат, возводя число в степень:

```
>>> math.pow(2, 3)
8.0
```

Python имеет также встроенный оператор экспоненты **, делающий то же самое, но он не преобразует автоматически результат к числу с плавающей точкой, если основание и степень были целыми числами:

```
>>> 2**3
8
>>> 2.0**3
8.0
```

Получаем квадратный корень с помощью функции sqrt():

```
>>> math.sqrt(100.0)
10.0
```

Не пытайтесь обмануть эту функцию, она уже все это видела:

```
>>> math.sqrt(-100.0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: math domain error
```

Здесь имеются также обычные тригонометрические функции, я просто приведу их названия: sin(), cos(), tan(), asin(), acos(), atan() и atan2(). Если вы помните теорему Пифагора (или хотя бы можете произнести ее название быстро три раза, не начав плеваться), библиотека math предоставит вам также функцию hypot(), которая рассчитывает гипотенузу на основании длины двух катетов:

```
>>> x = 3.0
>>> y = 4.0
>>> math.hypot(x, y)
5.0
```

Если вы не доверяете таким функциям, можете сделать все самостоятельно:

```
>>> math.sqrt(x*x + y*y)
5.0
>>> math.sqrt(x**2 + y**2)
5.0
```

Последний набор функций преобразует угловые координаты:

```
>>> math.radians(180.0)
3.141592653589793
>>> math.degrees(math.pi)
180.0
```

Работа с комплексными числами

Комплексные числа полностью поддерживаются основой языка Python в традиционной нотации с мнимой и действительной частями:

```
>>> # действительное число
... 5
5
>>> # мнимое число
... 8j
8j
>>> # мнимое число
... 3 + 2j
(3+2j)
```

Поскольку мнимое число i (в Python оно записывается как 1j) определено как квадратный корень из −1, мы можем выполнить следующее:

```
>>> 1j * 1j
(-1+0j)
>>> (7 + 1j) * 1j
(-1+7j)
```

Некоторые математические функции для комплексных чисел содержатся в стандартном модуле cmath.

Рассчитываем точное значение чисел с плавающей точкой с помощью decimal

Числа с плавающей точкой в вычислительной технике не похожи на настоящие числа, которые мы изучали в школе. Из-за того что компьютеры разрабатывались для бинарной математики, числа, не являющиеся равными степени двойки, зачастую не могут быть выражены точно:

```
>>> x = 10.0 / 3.0
>>> x
3.3333333333333335
```

Эй, что это за пятерка в конце? На ее месте должна быть тройка. С помощью модуля decimal (http://docs.python.org/3/library/decimal.html) вы можете записывать числа с желаемым уровнем точности. Это особенно важно для расчетов денежных сумм. Сумма в валюте Соединенных Штатов не может быть меньше 1 цента (сотая часть доллара), поэтому, если мы подсчитываем количество долларов и центов, нам нужно считать все до копеечки. Если мы попробуем представить доллары и центы как значения с плавающей точкой вроде 19,99 или 0,06, то потеряем некоторую часть точности в последних битах еще до начала вычислений. Как с этим справиться? Легко. Нам нужно использовать модуль decimal:

```
>>> from decimal import Decimal
>>> price = Decimal('19.99')
>>> tax = Decimal('0.06')
>>> total = price + (price * tax)
>>> total
Decimal('21.1894')
```

Мы записали цену и налог как строковые значения, чтобы сохранить их точность. При вычислении переменной total мы обработали все значащие части цента, но хотим получить ближайшее целое значение цента:

```
>>> penny = Decimal('0.01')
>>> total.quantize(penny)
Decimal('21.19')
```

Такие же результаты можно получить с помощью старых добрых чисел с плавающей точкой и округлений, но не всегда. Вы можете умножить сумму на 100 и использовать при подсчетах количество центов, но и при таком подходе можете пострадать. Этот вопрос обсуждается на сайте www.itmaybeahack.com.

Выполняем вычисления для рациональных чисел с помощью модуля fractions

Вы можете представлять числа как числитель, разделенный на знаменатель, с помощью модуля стандартной библиотеки `fractions` (http://docs.python.org/3/library/fractions.html). Рассмотрим пример простой операции умножения 1/3 на 2/3:

```
>>> from fractions import Fraction
>>> Fraction(1,  3) * Fraction(2, 3)
Fraction(2, 9)
```

Аргументы с плавающей точкой могут быть неточными, поэтому вы можете использовать модуль `Decimal` вместе с модулем `Fraction`:

```
>>> Fraction(1.0/3.0)
Fraction(6004799503160661, 18014398509481984)
>>> Fraction(Decimal('1.0')/Decimal('3.0'))
Fraction(3333333333333333333333333333, 10000000000000000000000000000)
```

Получим наибольший общий делитель для двух чисел с помощью функции `gcd`:

```
>>> import fractions
>>> fractions.gcd(24, 16)
8
```

Используем Packed Sequences с помощью array

В Python список больше похож на связанный список, а не на массив. Если вы хотите получить одномерную последовательность элементов одинакового типа, применяйте тип `array` (http://docs.python.org/3/library/array.html). Переменные этого типа используют меньше места и поддерживают многие методы работы со списками. Создайте массив с помощью команды вида `array` (*код типа, инициализатор*). Код типа указывает на тип данных (вроде `int` или `float`), и опциональный инициализатор содержит исходные значения, которые можно передать как список, строку или итерируемое значение.

Я никогда не использовал этот пакет для решения задач. Это низкоуровневая структура данных, полезная для представления чего-то вроде изображений. Если для выполнения числовых подсчетов вам на самом деле нужен массив, особенно имеющий больше одного измерения, лучше воспользоваться `NumPy`, который мы рассмотрим через пару разделов.

Обработка простой статистики с помощью модуля statistics

Начиная с версии Python 3.4, statistics является стандартным модулем (http://docs.python.org/3.4/library/statistics.html). Он содержит традиционные функции: среднее, медиану, моду, стандартное отклонение, распределение и т. д. Входными аргументами являются последовательности (списки или кортежи) или итераторы любого числового типа данных: `int`, `float`, `decimal` и `fraction`. Одна из функций, `mode`, также принимает в качестве аргументов строки. Для Python существует множество других статистических функций, располагающихся в пакетах SciPy и Pandas, которые мы рассмотрим далее в этом приложении.

Перемножение матриц

Начиная с Python 3.5, вы увидите символ `@`, который делает необычные вещи. Он все еще может использоваться для декораторов, но его также можно будет применять для перемножения матриц (http://legacy.python.org/dev/peps/pep-0465/). Однако до появления этой возможности вам следует воспользоваться `NumPy`.

Python для науки

В оставшейся части этого приложения рассматриваются сторонние пакеты Python для науки и математики. Несмотря на то что вы можете установить их индивидуально, следует рассмотреть их одновременную загрузку в качестве части научного дистрибутива Python. Рассмотрим основные варианты.

- Anaconda (https://store.continuum.io/cshop/anaconda/). Это бесплатный пакет, имеющий множество самых свежих возможностей. Он поддерживает Python 2 и 3 и не вредит установленной у вас версии Python.
- Python(x,y) (https://code.google.com/p/pythonxy/). Этот релиз подходит только для Windows.
- Pyzo (http://www.pyzo.org/). Этот пакет основан на некоторых инструментах пакета Anaconda, а также других.
- ALGORETE Loopy (http://algorete.org/). Этот пакет также основан на Anaconda и содержит дополнения.

Я рекомендую вам установить пакет Anaconda. Он большой, и все, что представлено в этом приложении, в нем содержится. В приложении Г рассматривается использование Python 3 и Anaconda. Примеры остальной части приложения подразумевают, что вы установили необходимые пакеты либо отдельно, либо как часть Anaconda.

NumPy

Это одна из основных причин популярности Python среди ученых (http://www.numpy.org/). Вы слышали, что динамические языки вроде Python зачастую медленнее компилирующих языков вроде C или даже других интерпретируемых языков вроде Java. NumPy был написан для предоставления доступа к быстрым многомерным массивам по аналогии с научными языками вроде FORTRAN. Вы получаете скорость C и дружественность к разработчикам Python.

Если вы загрузили один из научных дистрибутивов, у вас уже есть NumPy. Если же нет, следуйте инструкциям на странице загрузки NumPy (http://www.scipy.org/scipylib/download.html).

Для того чтобы начать работу с NumPy, вы должны понять устройство основной структуры данных, многомерного массива `ndarray` (от N-dimensional array — «N-мерный массив») или просто `array`. В отличие от списков и кортежей в Python, все элементы должны иметь одинаковый тип.

NumPy называет количество измерений массива его рангом. Одномерный массив похож на ряд значений, двухмерный — на таблицу с рядами и колонками, а трехмерный — на кубик Рубика. Длина измерений может не быть одинаковой.

array в NumPy и array в Python — это не одно и то же. В дальнейшем в этом приложении я буду работать только с массивами NumPy.

Но зачем нам нужны массивы?

- Научные данные зачастую представляют собой большие последовательности.
- Научные подсчеты для таких данных часто выполняются с использованием матричной математики, регрессии, симуляции и других приемов, которые обрабатывают множество фрагментов данных одновременно.
- NumPy обрабатывает массивы гораздо быстрее, чем стандартные списки или кортежи Python.

Существует множество способов создать массив NumPy.

Создание массива с помощью функции array()

Вы можете создать массив из обычного списка или кортежа:

```
>>> b = np.array([2, 4, 6, 8])
>>> b
array([2, 4, 6, 8])
```

Атрибут `ndim` возвращает ранг массива:

```
>>> b.ndim
1
```

Общее число значений можно получить с помощью атрибута `size`:

```
>>> b.size
4
```

Количество значений каждого ранга возвращает атрибут `shape`:

```
>>> b.shape
(4,)
```

Создание массива с помощью функции arange()

Метод `arange()` похож на стандартный метод `range()`. Если вы вызовете метод `arange()`, передав ему один целочисленный аргумент `num`, он вернет `ndarray` от `0` до `num-1`:

```
>>> import numpy as np
>>> a = np.arange(10)
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> a.ndim
1
>>> a.shape
(10,)
>>> a.size
10
```

С помощью двух значений он создаст массив от первого элемента до последнего минус один:

```
>>> a = np.arange(7, 11)
>>> a
array([ 7,  8,  9, 10])
```

Вы также можете передать как третий параметр размер шага, который будет использован вместо единицы:

```
>>> a = np.arange(7, 11, 2)
>>> a
array([7, 9])
```

До сих пор мы показывали примеры лишь с целыми числами, но метод `arrange()` работает и с числами с плавающей точкой:

```
>>> f = np.arange(2.0, 9.8, 0.3)
>>> f
array([ 2. ,  2.3,  2.6,  2.9,  3.2,  3.5,  3.8,  4.1,  4.4,  4.7,  5. ,
        5.3,  5.6,  5.9,  6.2,  6.5,  6.8,  7.1,  7.4,  7.7,  8. ,  8.3,
        8.6,  8.9,  9.2,  9.5,  9.8])
```

И последний прием: аргумент dtype указывает функции arrange(), какого типа значения следует создать:

```
>>> g = np.arange(10, 4, -1.5, dtype=np.float)
>>> g
array([ 10. ,   8.5,   7. ,   5.5])
```

Создание массива с помощью функций zeros(), ones() и random()

Метод zeros() возвращает массив, все значения которого равны 0. В эту функцию вам нужно передать аргумент, в котором будет указана желаемая форма массива. Так создается одномерный массив:

```
>>> a = np.zeros((3,))
>>> a
array([ 0.,  0.,  0.])
>>> a.ndim
1
>>> a.shape
(3,)
>>> a.size
3
```

Этот массив имеет ранг 2:

```
>>> b = np.zeros((2, 4))
>>> b
array([[ 0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.]])
>>> b.ndim
2
>>> b.shape
(2, 4)
>>> b.size
8
```

Другой особой функцией, заполняющей массив одинаковыми значениями, является ones():

```
>>> import numpy as np
>>> k = np.ones((3, 5))
>>> k
array([[ 1.,  1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.,  1.]])
```

Последняя функция создает массив и заполняет его случайными значениями из промежутка от 0,0 до 1,0:

```
>>> m = np.random.random((3, 5))
>>> m
```

```
array([[  1.92415699e-01,   4.43131404e-01,   7.99226773e-01,
          1.14301942e-01,   2.85383430e-04],
       [  6.53705749e-01,   7.48034559e-01,   4.49463241e-01,
          4.87906915e-01,   9.34341118e-01],
       [  9.47575562e-01,   2.21152583e-01,   2.49031209e-01,
          3.46190961e-01,   8.94842676e-01]])
```

Изменяем форму массива с помощью метода reshape()

До этого момента массив не особо отличался от списка или кортежа. Одним из различий между ними является возможность изменять его форму с помощью функции reshape():

```
>>> a = np.arange(10)
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> a = a.reshape(2, 5)
>>> a
array([[0, 1, 2, 3, 4],
       [5, 6, 7, 8, 9]])
>>> a.ndim
2
>>> a.shape
(2, 5)
>>> a.size

10
```

Вы можете изменять форму массива разными способами:

```
>>> a = a.reshape(5, 2)
>>> a
array([[0, 1],
       [2, 3],
       [4, 5],
       [6, 7],
       [8, 9]])
>>> a.ndim
2
>>> a.shape
(5, 2)
>>> a.size
10
```

Присваиваем кортеж, указывающий параметры формы, атрибуту shape:

```
>>> a.shape = (2, 5)
>>> a
array([[0, 1, 2, 3, 4],
       [5, 6, 7, 8, 9]])
```

Единственное ограничение — произведение рангов должно быть равным количеству значений (в нашем случае 10):

```
>>> a = a.reshape(3, 4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: total size of new array must be unchanged
```

Получаем элемент с помощью конструкции []

Одномерный массив работает как список:

```
>>> a = np.arange(10)
>>> a[7]
7
>>> a[-1]
9
```

Но если массив имеет другую форму, используйте индексы, разделенные запятыми:

```
>>> a.shape = (2, 5)
>>> a
array([[0, 1, 2, 3, 4],
       [5, 6, 7, 8, 9]])
>>> a[1,2]
7
```

Это отличается от двухмерного списка:

```
>>> l = [ [0, 1, 2, 3, 4], [5, 6, 7, 8, 9] ]
>>> l
[[0, 1, 2, 3, 4], [5, 6, 7, 8, 9]]
>>> l[1,2]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: list indices must be integers, not tuple
>>> l[1][2]
7
```

Еще один момент: разбиение работает, но опять же только внутри множества, заключенного в один набор квадратных скобок. Снова создадим привычный проверочный массив:

```
>>> a = np.arange(10)
>>> a = a.reshape(2, 5)
>>> a
array([[0, 1, 2, 3, 4],
       [5, 6, 7, 8, 9]])
```

Используйте разбиение, чтобы получить первый ряд — элементы начиная со смещения 2, до конца:

```
>>> a[0, 2:]
array([2, 3, 4])
```

Теперь получим последний ряд — все элементы вплоть до третьего с конца:

```
>>> a[-1, :3]
array([5, 6, 7])
```

Вы также можете присвоить значение более чем одному элементу с помощью разбиения. Следующее выражение присваивает значение 1000 колонкам (смещениям) 2 и 3 каждого ряда:

```
>>> a[:, 2:4] = 1000
>>> a
array([[   0,    1, 1000, 1000,    4],
       [   5,    6, 1000, 1000,    9]])
```

Математика массивов

Создание и изменение формы массивов так нас увлекли, что мы почти забыли сделать с ними что-то более полезное. Для начала используем переопределенный в NumPy оператор умножения (*), чтобы умножить все значения массива за раз:

```
>>> from numpy import *
>>> a = arange(4)
>>> a
array([0, 1, 2, 3])
>>> a *= 3
>>> a
array([0, 3, 6, 9])
```

Если вы пытались умножить каждый элемент обычного списка Python на число, вам бы понадобились цикл или включение:

```
>>> plain_list = list(range(4))
>>> plain_list
[0, 1, 2, 3]
>>> plain_list = [num * 3 for num in plain_list]
>>> plain_list
[0, 3, 6, 9]
```

Такое поведение применимо также к сложению, вычитанию, делению и другим функциям библиотеки NumPy. Например, вы можете инициализировать все элементы массива любым значением с помощью функции zeros() и оператора сложения:

```
>>> from numpy import *
>>> a = zeros((2, 5)) + 17.0
>>> a
array([[ 17.,  17.,  17.,  17.,  17.],
       [ 17.,  17.,  17.,  17.,  17.]])
```

Линейная алгебра

NumPy содержит множество функций линейной алгебры. Например, определим такую систему линейных уравнений:

```
4x + 5y = 20
 x + 2y = 13
```

Как мы можем найти x и y? Создадим два массива:

- коэффициенты (множители для x и y);
- зависимые переменные (правая часть уравнения):

```
>>> import numpy as np
>>> coefficients = np.array([ [4, 5], [1, 2] ])
>>> dependents = np.array([20, 13])
```

Теперь используем функцию solve() модуля linalg:

```
>>> answers = np.linalg.solve(coefficients, dependents)
>>> answers
array([ -8.33333333,  10.66666667])
```

В результате получим, что x примерно равен -8.3, а y примерно равен 10.6. Являются ли эти числа решениями уравнения?

```
>>> 4 * answers[0] + 5 * answers[1]
20.0
>>> 1 * answers[0] + 2 * answers[1]
13.0
```

Так и есть. Для того чтобы напечатать меньше текста, вы также можете указать NumPy найти скалярное произведение массивов:

```
>>> product = np.dot(coefficients, answers)
>>> product
array([ 20.,  13.])
```

Если решение верно, значения массива product должны быть близки к значениям массива dependents. Вы можете использовать функцию allclose(), чтобы проверить, являются ли массивы хотя бы приблизительно равными (они могут быть не полностью равными из-за округления чисел с плавающей точкой):

```
>>> np.allclose(product, dependents)
True
```

NumPy также имеет модули для работы с многочленами, преобразованиями Фурье, статистикой и распределением вероятностей.

Библиотека SciPy

Библиотека SciPy (http://www.scipy.org/) создана на основе NumPy и имеет даже больше функций. Релиз SciPy (http://www.scipy.org/scipylib/download.html) содержит NumPy, SciPy, Pandas (ее мы рассмотрим позже в этой главе) и другие библиотеки.

SciPy содержит множество модулей, включая те, которые выполняют следующие задачи:

- оптимизацию;
- ведение статистики;
- интерполяцию;
- линейную регрессию;
- интеграцию;
- обработку изображений;
- обработку сигналов.

Если вы уже работали с другими научными инструментами для компьютера, то обнаружите, что Python, NumPy и SciPy охватывают некоторые области, с которыми работает также коммерческий MatLab (http://www.mathworks.com/products/matlab/) или приложение с открытым исходным кодом R (http://www.r-project.org/).

Библиотека SciKit

Как и предыдущая библиотека, SciKit — это группа научных пакетов, построенная на основе SciPy. SciKit (https://scikits.appspot.com/scikits) специализируется на машинном обучении. Она поддерживает моделирование, классификацию, кластеризацию и разнообразные алгоритмы.

Библиотека IPython

Библиотека IPython (http://ipython.org/) стоит вашего времени по многим причинам. Вот некоторые из них.

- Наличие улучшенного интерактивного интерпретатора (альтернатива примерам с приглашением >>>, которые мы использовали на протяжении этой книги).
- Публикация кодов, диаграмм, текста и других медиа в веб-блокнотах.
- Поддержка параллельных вычислений (http://bit.ly/parallel-comp).

Рассмотрим интерпретатор и блокноты.

Лучший интерпретатор

Существуют разные версии IPython для Python 2 и Python 3, обе они устанавливаются Anaconda или другой современной научной сборкой Python. Используйте iPython 3 для версии Python 3:

```
$ ipython3
 Python 3.3.3 (v3.3.3:c3896275c0f6, Nov 16 2013, 23:39:35)
 Type "copyright", "credits" or "license" for more information.
 IPython 0.13.1 -An enhanced Interactive Python.
 ?         -> Introduction and overview of IPython's features.
 %quickref -> Quick reference.
 help      -> Python's own help system.
 object?   -> Details about 'object', use 'object??' for extra details.
 In [1]:
```

Стандартный интерпретатор Python использует приглашения >>> и ..., чтобы указать, где и когда вы должны вводить код. IPython отслеживает все, что вы вводите, в списке `In`, и все, что вы выводите, в списке `Out`. Каждый фрагмент входных данных может занимать больше одной строки, поэтому вам следует отправлять его, нажав клавишу **Shift**, пока держите нажатой клавишу **Enter**. Вот пример одной строки:

```
In [1]: print("Hello? World?")
Hello? World?
In [2]:
```

`In` и `Out` — это автоматически нумеруемые списки, которые позволяют вам получить доступ к любой введенной или выведенной информации.

Если вы введете символ ? после переменной, IPython укажет ее тип, значение, способы создания переменной этого типа и сообщит некоторую вспомогательную информацию:

```
In [4]: answer = 42
In [5]: answer?
Type:       int
String Form:42
Docstring:
int(x=0) -> integer
int(x, base=10) -> integer
Convert a number or string to an integer, or return 0 if no arguments
are given.  If x is a number, return x.__int__().  For floating point
numbers, this truncates towards zero.
If x is not a number or if base is given, then x must be a string,
bytes, or bytearray instance representing an integer literal in the
given base.  The literal can be preceded by '+' or '-' and be surrounded
by whitespace.  The base defaults to 10.  Valid bases are 0 and 2-36.
Base 0 means to interpret the base from the string as an integer literal.
>>> int('0b100', base=0)
4
```

Поиск по имени — это популярная особенность IDE вроде IPython. Если вы нажмете клавишу Tab после того, как введете несколько символов, IPython покажет все переменные, ключевые слова и функции, которые начинаются с этих символов. Определим несколько переменных, а затем найдем все, что начинается с буквы «f»:

```
In [6]: fee = 1
In [7]: fie = 2
In [8]: fo = 3
In [9]: fum = 4
In [10]: ftab
%%file    fie       finally   fo        format    frozenset
fee       filter    float     for       from      fum
```

Если вы введете `fe` и нажмете клавишу Tab, то увидите на экране переменную `fee` — единственную в этой программе, начинающуюся с буквосочетания `fe`:

```
In [11]: fee
Out[11]: 1
```

Блокноты IPython

Если вы предпочитаете графические интерфейсы, вам может понравиться веб-интерфейс IPython. Вы начинаете из окна запуска Anaconda (рис. В.1).

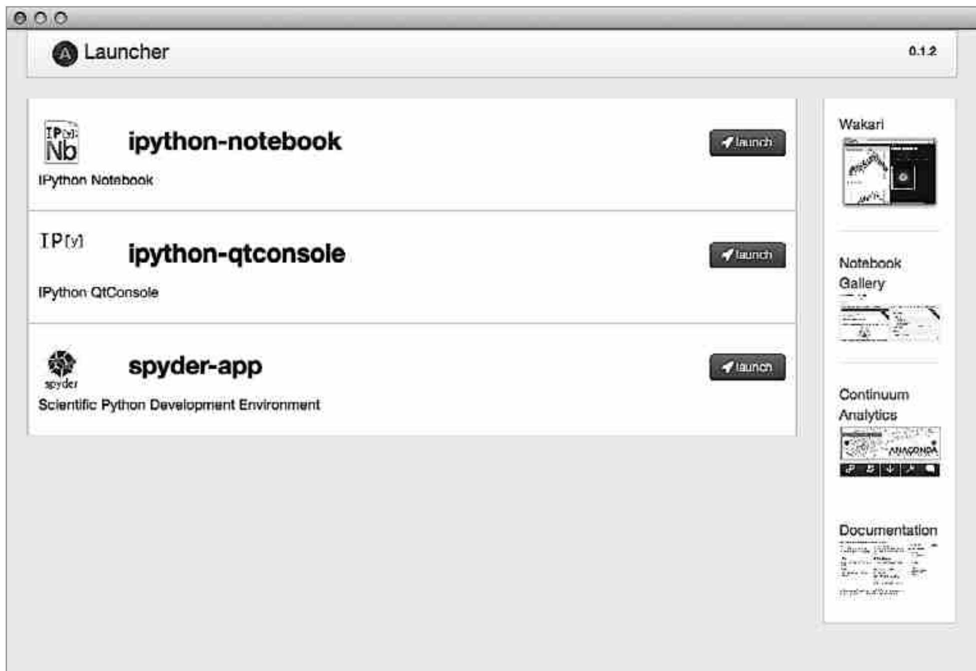


Рис. В.1. Домашняя страница Anaconda

Для того чтобы запустить блокнот в браузере, щелкните на значке Launch (Запустить), расположенный справа от строки ipython-notebook. На рис. В.2 показан начальный экран.

Рис. В.2. Домашняя страница IPython

Теперь нажмем кнопку New Notebook (Новый блокнот). Появится окно, похожее на то, что показано на рис. В.3.

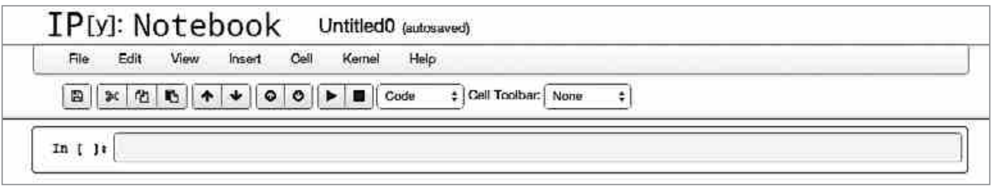


Рис. В.3. Страница блокнота IPython

Для графической версии нашего предыдущего примера, основанного на тексте, введите ту же команду, которую мы использовали в предыдущем разделе, как показано на рис. В.4.

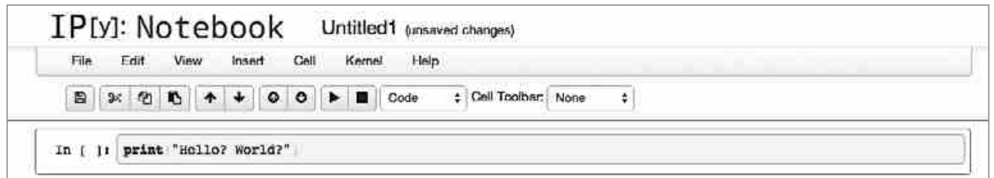


Рис. В.4. Вводим код в IPython

Нажмите на черный треугольный значок, чтобы запустить код. Результат показан на рис. В.5.

Такой блокнот — это не только графическая версия улучшенного интерпретатора. Помимо кода, он может содержать текст и форматированные математические выражения.

Рис. В.5. Запускаем код в IPython

В ряду значков в верхней части блокнота есть раскрывающееся меню (рис. В.6), с помощью которого вы можете указать, как хотите вводить содержимое. Можно выбрать один из следующих вариантов:

- код — стандартный вариант для кода Python;
- разметка — альтернатива HTML, которая служит для отображения отформатированного читабельного текста;
- простой текст — неформатированный текст от Heading 1 (Заголовок 1) до Heading 6 (Заголовок 6) — теги HTML от <H1> до <H6>.

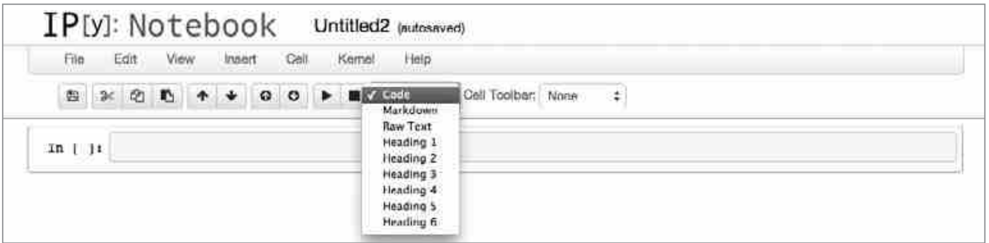


Рис. В.6. Меню выбора содержимого

Смешаем текст с кодом, сделав некое подобие «Википедии». Выберите пункт Heading 1 из раскрывающегося меню, введите `Humble Brag Example`, а затем нажмите клавишу Shift и, удерживая ее, клавишу Enter. Вы должны увидеть эти три слова, выделенные крупным полужирным шрифтом. Далее в раскрывающемся меню выберите пункт Code и введите такой код:

```
print("Some people say this code is ingenious")
```

Затем снова нажмите Shift+Enter. Вы должны увидеть отформатированный заголовок и код, как показано на рис. В.7.

Объединяя код, выходную информацию, текст и даже изображения, вы можете создать интерактивный блокнот. Поскольку доступ к нему можно получить по сети, он будет доступен из любого браузера.

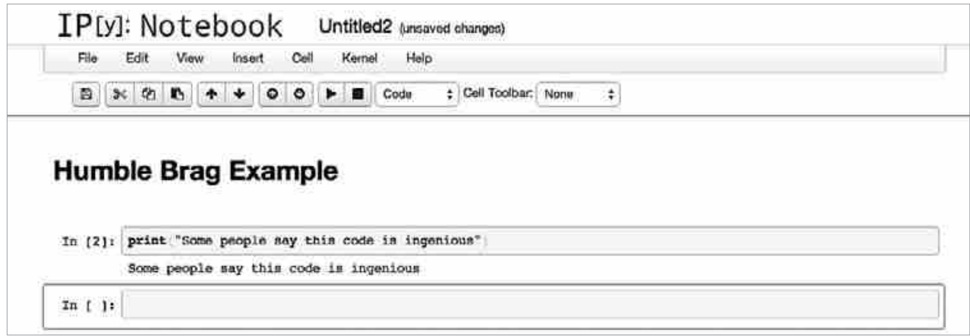


Рис. В.7. Форматированный текст и код

Вы можете увидеть блокноты, преобразованные в статический HTML (http://nbviewer.ipython.org/) или в галерею (http://bit.ly/ipy-notebooks). Например, взгляните на блокнот о пассажирах «Титаника» (http://bit.ly/titanic-noteb). Он содержит таблицы, в которых показывается, как пол, благосостояние и местонахождение на корабле повлияли на выживание. В качестве бонуса можете прочитать, как использовать различные технологии машинного обучения.

Ученые начинают применять блокноты IPython для того, чтобы публиковать свои исследования, включая весь код и данные, использованные в них.

Pandas

С недавнего времени распространено употребление словосочетания «наука о данных». Я слышал определения «статистика, собираемая на Mac» или «статистика, собираемая в Сан-Франциско». Как бы вы ни определили ее, инструменты, о которых я говорил ранее, — NumPy, SciPy и инструменты Pandas, вынесенные в тему этого раздела, — это компоненты растущего популярного инструментального средства, работающего с данными. (Mac и Сан-Франциско опциональны.)

Pandas — это новый пакет для интерактивного анализа данных (http://pandas.pydata.org/). Он особенно полезен для манипулирования данными реального мира с помощью комбинирования матричной математики NumPy и возможности обработки таблиц и реляционных баз данных. В книге Веса Маккинни (Wes McKinney) *Python for Data Analysis: Data Wrangling with Pandas, NumPy, and IPython* (издательство O'Reilly) рассматриваются выпас данных с помощью NumPy, Python и Pandas.

NumPy ориентирован на традиционные научные вычисления, которые, как правило, манипулируют многомерными множествами данных одного типа, обычно числами с плавающей точкой. Pandas больше похож на редактор базы данных, обрабатывающий несколько типов данных в группе. В некоторых языках такие группы называются записями или структурами. Pandas определяет базовую структуру данных, которая называется DataFrame. Она представляет собой упорядоченную коллекцию граф с именами и типами и напоминает таблицу, именованный кортеж или вложенный словарь в Python. Ее предназначение заключается в упрощении

работы с любыми данными, которые вы можете встретить не только в науке, но и в бизнесе. Фактически Pandas разрабатывался для работы с финансовыми данными, наиболее распространенной альтернативой для которых является электронная таблица.

Pandas — это ETL-инструмент для реальных данных — с отсутствующими значениями, странными форматами, странными измерениями — всех типов. Вы можете разделить, объединить, заполнить, сконвертировать, изменить форму, разбить данные, а также загрузить и сохранить файлы. Он интегрируется с инструментами, которые мы только что обсудили, — NumPy, SciPy, iPython — для подсчета статистики, подгонки данных под модель, рисования диаграмм, публикации и т. д.

Большинство ученых хотят выполнять свою работу, не тратя месяцы на то, чтобы стать экспертами в эзотерических языках программирования или приложениях. С помощью Python они быстрее могут стать более продуктивными.

Python и научные области

Мы рассматривали инструменты Python, которые могут быть использованы практически в любой области науки. Но как насчет программного обеспечения и документации, нацеленных на конкретные научные области? Рассмотрим примеры использования Python для решения определенных задач и некоторые узконаправленные библиотеки.

- Общие:
  - вычисления Python в науке и инженерном деле (http://bit.ly/py-comp-sci);
  - интенсивный курс Python для ученых (http://bit.ly/pyforsci).
- Физика — физические вычисления (http://bit.ly/pyforsci).
- Биология и медицина:
  - Python для биологов (http://pythonforbiologists.com/);
  - Neuroimaging с помощью Python (http://nipy.org/).

Проводятся следующие международные конференции по Python и научным данным:

- PyData (http://pydata.org/);
- SciPy (http://conference.scipy.org/);
- EuroSciPy (https://www.euroscipy.org/).

Г Установка Python 3

К моменту, когда Python 3 будет предустановлен на каждом компьютере, тостеры будут заменены 3D-принтерами, которые каждый день будут выдавать пончики. В операционной системе Windows вообще нет Python, а OS X, Linux и Unix, как правило, имеют старые версии. До тех пор пока это не исправили, вам, скорее всего, придется устанавливать Python 3 самостоятельно.

Далее показывается, как выполнить следующие задачи:

- определить, какая версия Python установлена на вашем компьютере, если она есть;
- установить стандартный дистрибутив Python 3, если у вас его нет;
- установить дистрибутив Anaconda, содержащий научные модули Python;
- установить pip и virtualenv, если вы не можете изменять свою систему;
- установить conda в качестве альтернативы pip.

Большинство примеров этой книги были написаны и протестированы для Python 3.3, последней стабильной версии на момент ее написания. В некоторых примерах использовалась версия 3.4, которая была выпущена в момент, когда книга редактировалась. Страница What's New in Python (https://docs.python.org/3/whatsnew/) представляет информацию о том, что было добавлено в каждой версии. Существует множество исходных кодов Python и много способов установить новую версию. В этом приложении я опишу два из них.

- Если вы хотите установить стандартный интерпретатор и библиотеки, я рекомендую вам посетить официальный сайт языка (http://www.python.org/).
- Если вы хотите использовать и стандартную библиотеку, и научные библиотеки, описанные в приложении В, используйте Anaconda.

Установка стандартной версии Python

Перейдите в браузере на страницу загрузки Python (http://www.python.org/download/). Она попробует определить вашу операционную систему и предоставить подхо-

дящие вам варианты. Если она ошибется, вы можете использовать следующие ссылки:

- версии Python для Windows (https://www.python.org/downloads/windows/);
- версии Python для Mac OS X (https://www.python.org/downloads/mac-osx/);
- исходные коды Python (Linux и Unix) (https://www.python.org/downloads/source/).

Вы увидите страницу, похожую на ту, что показана на рис. Г.1.

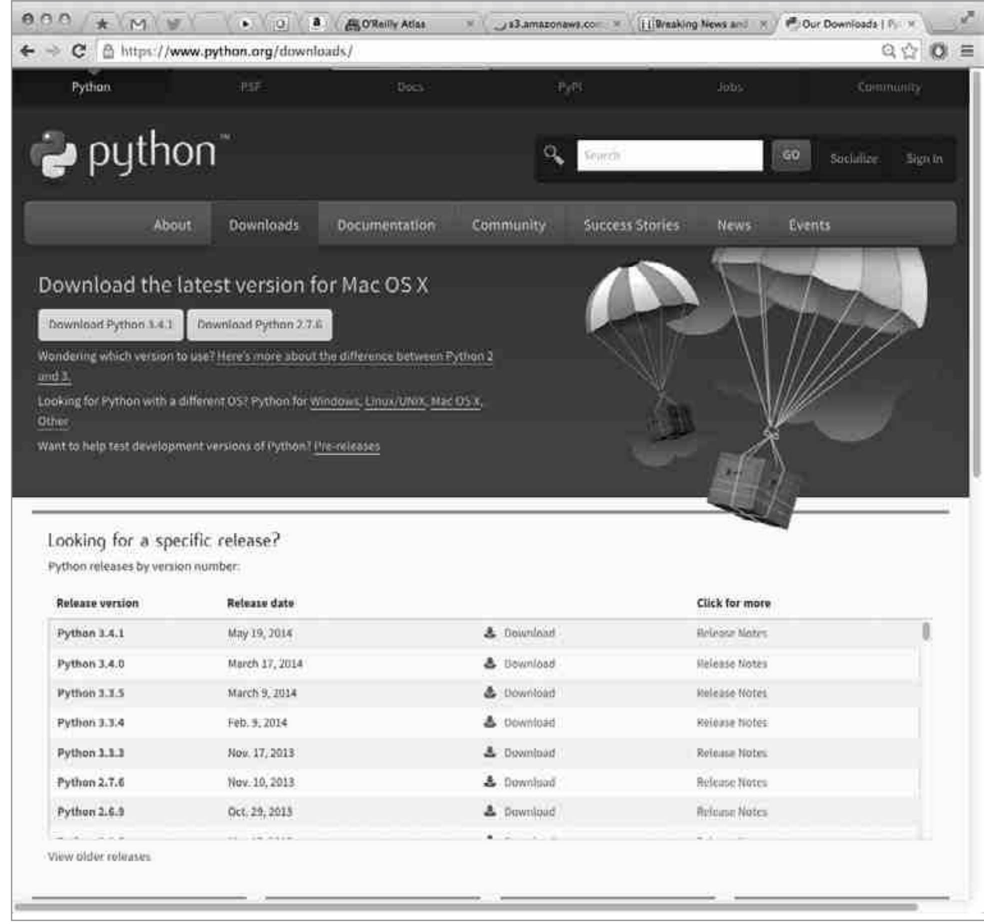


Рис. Г.1. Пример страницы загрузки

Выберите ссылку Download (Загрузить) у наиболее свежей версии. В нашем случае это 3.4.1. Это отправит вас на страницу информации, похожую на ту, что показана на рис. Г.2.

Вам нужно прокрутить страницу вниз, чтобы увидеть ссылку для загрузки (рис. Г.3).

Рис. Г.2. Страница деталей загрузки

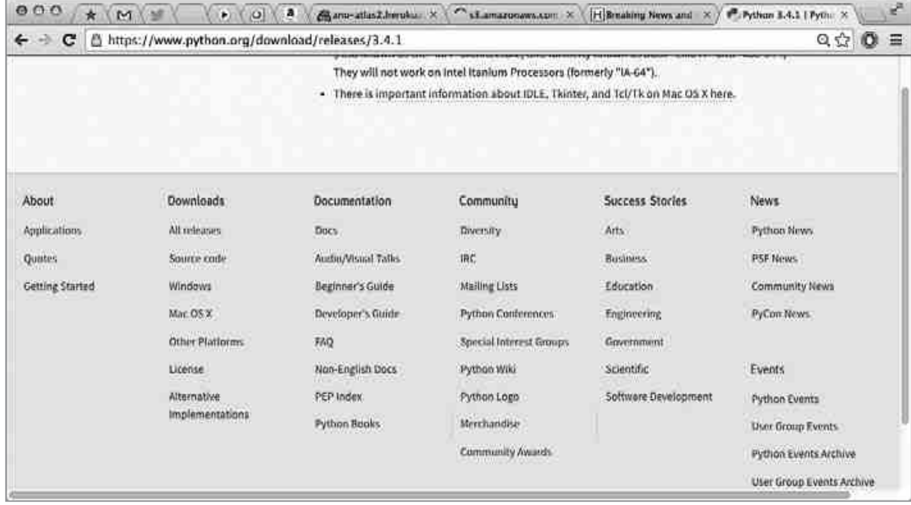


Рис. Г.3. Нижняя часть страницы, предлагающая загрузить Python

Выберите ссылку, чтобы перейти на страницу определенной версии (рис. Г.4). Теперь выберите корректную версию для вашего компьютера.

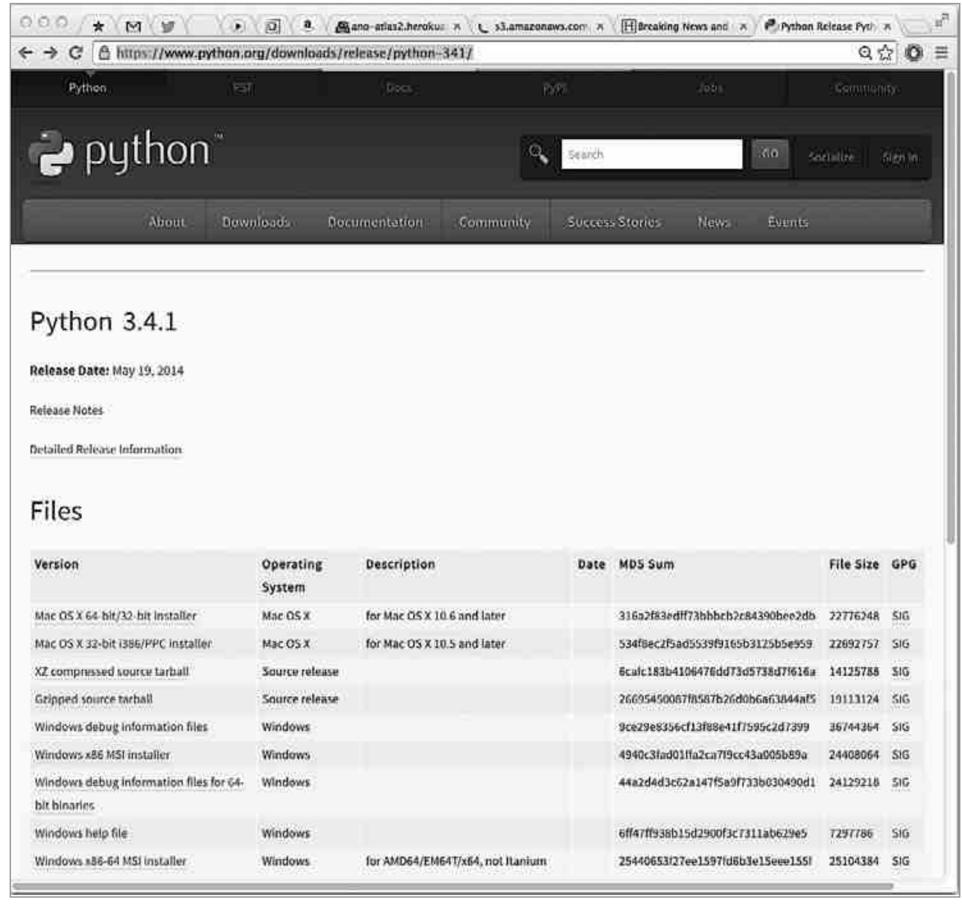


Рис. Г.4. Файлы для загрузки

Mac OS X

Щелкните на ссылке Mac OS X 64-bit/32-bit installer, чтобы загрузить файл с расширением `.dmg` для Mac. После завершения загрузки дважды щелкните на нем. Появится окно с четырьмя значками. Правой кнопкой мыши щелкните на Python.mpkg и затем в появившемся диалоговом окне нажмите кнопку Open (Открыть). Нажмите кнопку Continue (Продолжить) три раза (или около того), чтобы просмотреть юридические детали, и затем, когда появится соответствующее диалоговое окно, нажмите кнопку Install (Установить). Python 3 будет установлен в каталог `/usr/local/bin/python3`, что оставит существующую версию Python 2 нетронутой.

Windows

Для Windows загрузите один из следующих установщиков:

- Windows x86 MSI installer (32-bit) (http://bit.ly/win-x86);
- Windows x86-64 MSI installer (64-bit) (http://bit.ly/win-x86-64).

Чтобы определить, какая версия Windows у вас установлена (32- или 64-битная), сделайте следующее.

1. Нажмите кнопку Пуск.
2. Щелкните правой кнопкой мыши на пункте Мой компьютер.
3. Выберите пункт меню Свойства и найдите битовое значение.

Щелкните на соответствующем установщике (файл с расширением .msi). После того как он будет загружен, щелкните на нем два раза и следуйте инструкциям.

Linux или Unix

Пользователи Linux и Unix могут выбрать формат сжатия файлов исходного кода:

- сжатие с помощью XZ (http://bit.ly/xz-tarball);
- сжатие с помощью Gzipped (http://bit.ly/gzip-tarball).

Загружайте любой из этих архивов. Разархивируйте его с помощью tar xJ (для файла с расширением .xz) или tar xz (для файла с расширением .tgz), а затем запустите полученный сценарий оболочки.

Установка Anaconda

Anaconda — это всеобъемлющий установщик с акцентом на науку: он содержит Python, стандартную библиотеку и множество полезных сторонних библиотек. До недавнего момента он содержал Python 2 в качестве стандартного интерпретатора, несмотря на то что существовала возможность установить Python 3.

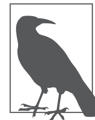
Новая версия, Anaconda 2.0, устанавливает последнюю версию Python и ее стандартную библиотеку (3.4 на момент написания этой книги). Среди других прелестей — библиотеки, о которых мы говорили ранее в этой книге: beautifulsoup4, Flask, ipython, matplotlib, nose, numpy, Pandas, pillow, pip, scipy, tables, zmq и множество других. Он содержит кросс-платформенную программу установки, которая называется conda, она улучшает pip4 — мы поговорим об этом через некоторое время.

Чтобы установить Anaconda 2, перейдите на страницу загрузки версий Python 3 (http://repo.continuum.io/anaconda3/). Нажмите соответствующую ссылку для вашей платформы (номера версий могут измениться с момента написания этой книги, но вы сможете с этим разобраться).

- Для того чтобы загрузить версию для Mac, нажмите ссылку Anaconda3-2.0.0-MacOSX-x86_64.pkg. После загрузки щелкните на файле два раза и сделайте

обычные шаги установки ПО для Mac. Все содержимое будет установлено в папку anaconda, расположенную в вашем домашнем каталоге.

- Для Windows выберите 32- или 64-битную версию. После загрузки дважды щелкните на файле с расширением .exe.
- Для Linux выберите 32- или 64-битную версию. После загрузки запустите его (это большой сценарий оболочки).



Убедитесь, что имя загружаемого вами файла начинается на Anaconda3. Если оно начинается с Anaconda, это версия для Python 2.

Anaconda устанавливает содержимое в собственную папку (она называется anaconda и располагается в домашнем каталоге). Это значит, что он не будет мешать другим версиям Python, установленным на ваш компьютер. Это также значит, что вам не нужно особое разрешение (иметь имя вроде admin или root) для того, чтобы его установить.

Чтобы увидеть, какие пакеты содержит Anaconda, посетите страницу документации (http://docs.continuum.io/anaconda/pkg-docs.html) и затем, когда в верхней части страницы появится список, выберите Python version: 3.4 (Python, версия 3.4). Когда я проверял в последний раз, я увидел 141 пакет.

После установки Anaconda 2 вы сможете увидеть, что Санта положил в ваш компьютер, введя эту команду:

```
$ ./conda list
# packages in environment at /Users/williamlubanovic/anaconda:
#
anaconda                  2.0.0              np18py34_0
argcomplete               0.6.7                  py34_0
astropy                   0.3.2              np18py34_0
backports.ssl-match-hostname 3.4.0.2                <pip>
beautiful-soup            4.3.1                  py34_0
beautifulsoup4            4.3.1                   <pip>
binstar                   0.5.3                  py34_0
bitarray                  0.8.1                  py34_0
blaze                     0.5.0              np18py34_0
blz                       0.6.2              np18py34_0
bokeh                     0.4.4              np18py34_1
cdecimal                  2.3                    py34_0
colorama                  0.2.7                  py34_0
conda                     3.5.2                  py34_0
conda-build               1.3.3                  py34_0
configobj                 5.0.5                  py34_0
curl                      7.30.0                      2
cython                    0.20.1                 py34_0
datashape                 0.2.0              np18py34_1
```

```
dateutil                  2.1                  py34_2
docutils                  0.11                 py34_0
dynd-python               0.6.2            np18py34_0
flask                     0.10.1               py34_1
freetype                  2.4.10                    1
future                    0.12.1               py34_0
greenlet                  0.4.2                py34_0
h5py                      2.3.0            np18py34_0
hdf5                      1.8.9                     2
ipython                   2.1.0                py34_0
ipython-notebook          2.1.0                py34_0
ipython-qtconsole         2.1.0                py34_0
itsdangerous              0.24                 py34_0
jdcal                     1.0                  py34_0
jinja2                    2.7.2                py34_0
jpeg                      8d                        1
libdynd                   0.6.2                     0
libpng                    1.5.13                    1
libsodium                 0.4.5                     0
libtiff                   4.0.2                     0
libxml2                   2.9.0                     1
libxslt                   1.1.28                    2
llvm                      3.3                       0
llvmpy                    0.12.4               py34_0
lxml                      3.3.5                py34_0
markupsafe                0.18                 py34_0
matplotlib                1.3.1            np18py34_1
mock                      1.0.1                py34_0
multipledispatch          0.4.3                py34_0
networkx                  1.8.1                py34_0
nose                      1.3.3                py34_0
numba                     0.13.1           np18py34_0
numexpr                   2.3.1            np18py34_0
numpy                     1.8.1                py34_0
openpyxl                  2.0.2                py34_0
openssl                   1.0.1g                    0
pandas                    0.13.1           np18py34_0
patsy                     0.2.1            np18py34_0
pillow                    2.4.0                py34_0
pip                       1.5.6                py34_0
ply                       3.4                  py34_0
psutil                    2.1.1                py34_0
py                        1.4.20               py34_0
pycosat                   0.6.1                py34_0
pycparser                 2.10                 py34_0
pycrypto                  2.6.1                py34_0
pyflakes                  0.8.1                py34_0
pygments                  1.6                  py34_0
```

```
pyparsing                 2.0.1                  py34_0
pyqt                      4.10.4                 py34_0
pytables                  3.1.1              np18py34_0
pytest                    2.5.2                  py34_0
python                    3.4.1                       0
python-dateutil           2.1                     <pip>
python.app                1.2                    py34_2
pytz                      2014.3                 py34_0
pyyaml                    3.11                   py34_0
pyzmq                     14.3.0                 py34_0
qt                        4.8.5                       3
readline                  6.2                         2
redis                     2.6.9                       0
redis-py                  2.9.1                  py34_0
requests                  2.3.0                  py34_0
rope                      0.9.4                  py34_1
rope-py3k                 0.9.4                   <pip>
runipy                    0.1.0                  py34_0
scikit-image              0.9.3              np18py34_0
scipy                     0.14.0             np18py34_0
setuptools                3.6                    py34_0
sip                       4.15.5                 py34_0
six                       1.6.1                  py34_0
sphinx                    1.2.2                  py34_0
spyder                    2.3.0rc1               py34_0
spyder-app                2.3.0rc1               py34_0
sqlalchemy                0.9.4                  py34_0
sqlite                    3.8.4.1                     0
ssl_match_hostname        3.4.0.2                py34_0
sympy                     0.7.5                  py34_0
tables                    3.1.1                   <pip>
tk                        8.5.15                      0
tornado                   3.2.1                  py34_0
ujson                     1.33                   py34_0
werkzeug                  0.9.4                  py34_0
xlrd                      0.9.3                  py34_0
xlsxwriter                0.5.5                  py34_0
yaml                      0.1.4                       1
zeromq                    4.0.4                       0
zlib                      1.2.7                       1
```

Установка и использование pip и virtualenv

Пакет pip — это самый популярный способ установить сторонние (нестандартные) пакеты Python. Несколько раздражает то, что такой полезный инструмент не являлся частью стандартного Python и его приходилось загружать и устанавливать самостоятельно. Как говорил мой друг, это жестокий, пугающий ритуал. Хорошая

новость заключается в том, что, начиная с версии 3.4, pip является стандартной частью Python.

Вместе с pip часто используется программа virtualenv — это способ установить пакеты Python в заданный каталог, чтобы избежать взаимодействий с уже существующими пакетами Python. Это позволяет вам использовать любые Python-функции, даже если у вас нет разрешения изменять текущую установленную версию.

Если у вас установлен Python 3, но под рукой только версия pip для Python 2, получить версию для Python 3 под Linux или OS X можно следующим способом:

```
$ curl -O http://python-distribute.org/distribute_setup.py
$ sudo python3 distribute_setup.py
$ curl -O https://raw.github.com/pypa/pip/master/contrib/get-pip.py
$ sudo python3 get-pip.py
```

Это установит pip-3.3 в каталог bin вашей версии Python 3. Далее для установки сторонних пакетов вы можете использовать pip-3.3 вместо версии для Python 2.

Вот несколько хороших руководств по pip и virtualenv:

- http://bit.ly/jm-pip-vlenv;
- http://bit.ly/hhgp-pip.

Установка и использование conda

До недавнего момента pip всегда загружал файлы исходных кодов, а не бинарные файлы. Это могло стать проблемой для модулей Python, которые построены на основе библиотек C. Недавно разработчики Anaconda создали conda (http://www.continuum.io/blog/conda), для того чтобы решить эту проблему. pip — это менеджер пакетов для Python, а conda работает с любыми языками программирования и ПО. conda также не нуждается в чем-то вроде virtualenv для того, чтобы содержать отдельно разные пакеты.

Если вы установили дистрибутив Anaconda, у вас уже есть программа conda. Если нет, можете установить Python 3 и conda со страницы http://conda.pydata.org/miniconda.html. Как и в случае с Anaconda, убедитесь, что файл, который вы загружаете, начинается с Miniconda3, а не с Miniconda — это версия для Python 2.

conda работает вместе с pip. Несмотря на то что он имеет собственный публичный репозиторий пакетов (http://binstar.org/), команды вроде `conda search` также выполнят поиск в репозитории PyPI (http://pypi.python.org/). Если у вас возникают проблемы с pip, conda может стать хорошей альтернативой.

Д Ответы к упражнениям

Глава 1. Python: с чем его едят

1. Если вы еще не установили Python 3, сделайте это сейчас. Прочтите приложение Г, чтобы узнать детали.
2. Запустите интерактивный интерпретатор Python 3. И вновь детали вы найдете в приложении Г. Интерпретатор должен вывести несколько строк о себе, а затем строку, начинающуюся с символов >>>. Перед вами приглашение для ввода команд Python.

   Вот так это выглядит на моем MacBook Pro:

```
$ python
Python 3.3.0 (v3.3.0:bd8afb90ebf2, Sep 29 2012, 01:25:11)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

3. Немного поэкспериментируйте с интерпретатором. Используйте его как калькулятор и наберите текст `8 * 9`. Нажмите клавишу Enter, чтобы увидеть результат. Python должен вывести `72`:

```
>>> 8 * 9
72
```

4. Теперь введите число `47` и нажмите Enter. Появилось ли число `47` в следующей строке?

```
>>> 47
47
```

5. Теперь введите `print(47)` и нажмите клавишу Enter. Появилось ли снова число `47` в следующей строке?

```
>>> print(47)
47
```

Глава 2. Ингредиенты Python: числа, строки и переменные

1. Сколько секунд содержится в часе? Используйте интерактивный интерпретатор как калькулятор и умножьте количество секунд в минуте (60) на количество минут в часе (тоже 60):

```
>>> 60 * 60
3600
```

2. Присвойте результат вычисления предыдущего задания (секунды в часе) переменной, которая называется seconds_per_hour:

```
>>> seconds_per_hour = 60 * 60
>>> seconds_per_hour
3600
```

3. Сколько секунд содержится в сутках? Используйте переменную seconds_per_hour:

```
>>> seconds_per_hour * 24
86400
```

4. Снова посчитайте количество секунд в сутках, но на этот раз сохраните результат в переменной seconds_per_day:

```
>>> seconds_per_day = seconds_per_hour * 24
>>> seconds_per_day
86400
```

5. Разделите значение переменной seconds_per_day на значение переменной seconds_per_hour. Используйте деление с плавающей точкой (/):

```
>>> seconds_per_day / seconds_per_hour
24.0
```

6. Разделите значение переменной seconds_per_day на значение переменной seconds_per_hour. Используйте целочисленное деление (//). Совпадает ли полученный результат с ответом на предыдущее упражнение, если не учитывать символы .0 в конце?

```
>>> seconds_per_day // seconds_per_hour
24
```

Глава 3. Наполнение Python: списки, кортежи, словари и множества

1. Создайте список, который называется years_list, содержащий год, в который вы родились, и каждый последующий год вплоть до вашего пятого дня рождения.

Например, если вы родились в 1980 году, список будет выглядеть так: `years_list = [1980, 1981, 1982, 1983, 1984, 1985]`.

```
Если вы родились в 1980, вам нужно ввести следующее:
>>> years_list = [1980, 1981, 1982, 1983, 1984, 1985]
```

2. В какой из годов, содержащихся в списке `years_list`, был ваш третий день рождения? Помните, в первый год вам было 0 лет.

   Вам нужно смещение 3. Поэтому, если вы родились в 1980-м:

```
>>> years_list[3]
1983
```

3. В какой из годов, содержащихся в списке `years_list`, вам было больше всего лет?

   Вам нужно получить последний год, поэтому используйте смещение `-1`. Вы также можете использовать смещение `5`, поскольку знаете, что в этом списке всего шесть элементов. Однако смещение `-1` позволяет получить последний элемент из списка любой длины. Для тех, кто родился в 1980 году:

```
>>> years_list[-1]
1985
```

4. Создайте список `things`, содержащий три элемента: `"mozzarella"`, `"cinderella"`, `"salmonella"`:

```
>>> things = ["mozzarella", "cinderella", "salmonella"]
>>> things
['mozzarella', 'cinderella', 'salmonella']
```

5. Напишите с большой буквы тот элемент списка `things`, который относится к человеку, а затем выведите список. Изменился ли элемент списка?

   Эта строка записывает слово с прописной буквы, но не меняет его в списке:

```
>>> things[1].capitalize()
'Cinderella'
>>> things
['mozzarella', 'cinderella', 'salmonella']
```

   Если вы хотите изменить его в списке, вам нужно присвоить его снова:

```
>>> things[1] = things[1].capitalize()
>>> things
['mozzarella', 'Cinderella', 'salmonella']
```

6. Переведите сырный элемент списка `things` в верхний регистр целиком и выведите список:

```
>>> things[0] = things[0].upper()
>>> things
['MOZZARELLA', 'Cinderella', 'salmonella']
```

7. Удалите болезнь из списка `things`, получите Нобелевскую премию и затем выведите список на экран.

   Это удалит элемент по значению:

```
>>> things.remove("salmonella")
>>> things
['MOZZARELLA', 'Cinderella']
```

   Поскольку элемент находится на последнем месте в списке, следующая строка тоже сработает:

```
>>> del things[-1]
```

   Элемент также можно удалить, указав смещение от начала:

```
>>> del things[2]
```

8. Создайте список, который называется `surprise` и содержит элементы `'Groucho'`, `'Chico'` и `'Harpo'`.

```
>>> surprise = ['Groucho', 'Chico', 'Harpo']
>>> surprise
['Groucho', 'Chico', 'Harpo']
```

9. Напишите последний элемент списка `surprise` со строчной буквы, затем обратите его и напишите с прописной буквы:

```
>>> surprise[-1] = surprise[-1].lower()
>>> surprise[-1] = surprise[-1][::-1]
>>> surprise[-1].capitalize()
'Oprah'
```

10. Создайте англо-французский словарь, который называется `e2f`, и выведите его на экран. Вот ваши первые слова: dog/chien, cat/chat и walrus/morse:

```
>>> e2f = {'dog': 'chien', 'cat': 'chat', 'walrus': 'morse'}
>>> e2f
{'cat': 'chat', 'walrus': 'morse', 'dog': 'chien'}
```

11. Используя словарь `e2f`, выведите французский вариант слова walrus:

```
>>> e2f['walrus']
'morse'
```

12. Создайте французско-английский словарь `f2e` на основе словаря `e2f`. Используйте метод `items`:

```
>>> f2e = {}
>>> for english, french in e2f.items():
    f2e[french] = english
>>> f2e
{'morse': 'walrus', 'chien': 'dog', 'chat': 'cat'}
```

13. Используя словарь f2e, выведите английский вариант слова chien:

```
>>> f2e['chien']
'dog'
```

14. Создайте и выведите на экран множество английских слов из ключей словаря e2f:

```
>>> set(e2f.keys())
{'cat', 'walrus', 'dog'}
```

15. Создайте многоуровневый словарь life. Используйте следующие строки для ключей верхнего уровня: 'animals', 'plants' и 'other'. Сделайте так, чтобы ключ 'animals' ссылался на другой словарь, имеющий ключи 'cats', 'octopi' и 'emus'. Сделайте так, чтобы ключ 'cats' ссылался на список строк со значениями 'Henri', 'Grumpy' и 'Lucy'. Остальные ключи должны ссылаться на пустые словари.

Это довольно трудный пример, поэтому, если вы подглядели сюда, ничего особо страшного не случилось:

```
>>> life = {
...     'animals': {
...         'cats': [
...             'Henri', 'Grumpy', 'Lucy'
...             ],
...         'octopi': {},
...         'emus': {}
...         },
...     'plants': {},
...     'other': {}
...     }
>>>
```

16. Выведите на экран высокоуровневые ключи словаря life:

```
>>> print(life.keys())
dict_keys(['animals', 'other', 'plants'])
```

Python 3 содержит функционал для работы с ключами словарей. Чтобы вывести их как список, используйте следующую строку:

```
>>> print(list(life.keys()))
['animals', 'other', 'plants']
```

Вы можете использовать пробелы, чтобы сделать ваш код более удобочитаемым:

```
>>> print (list (life.keys()))
['animals', 'other', 'plants']
```

17. Выведите на экран ключи `life['animals']`:

```
>>> print(life['animals'].keys())
dict_keys(['cats', 'octopi', 'emus'])
```

18. Выведите значения `life['animals']['cats']`:

```
>>> print(life['animals']['cats'])
['Henri', 'Grumpy', 'Lucy']
```

Глава 4. Корочка Python: структуры кода

1. Присвойте значение `7` переменной `guess_me`. Далее напишите условные проверки (`if`, `else` и `elif`), чтобы вывести строку `'too low'`, если значение переменной `guess_me` меньше `7`, `'too high'` — если оно больше `7`, и `'just right'` — если равно `7`:

```
guess_me = 7
if guess_me < 7:
    print('too low')
elif guess_me > 7:
    print('too high')
else:
    print('just right')
```

Запустите эту программу, и вы увидите следующую строку:

```
just right
```

2. Присвойте значение `7` переменной `guess_me` и значение `1` переменной `start`. Напишите цикл `while`, который сравнивает переменные `start` и `guess_me`. Выведите строку `'too low'`, если значение переменной `start` меньше значения переменной `guess_me`. Если значение переменной `start` равно значению переменной `guess_me`, выведите строку `'found it!'` и выйдите из цикла. Если значение переменной `start` больше значения переменной `guess_me`, выведите строку `'oops'` и выйдите из цикла. Увеличьте значение переменной `start` на выходе из цикла:

```
guess_me = 7
start = 1
while True:
    if start < guess_me:
        print('too low')
    elif start == guess_me:
        print('found it!')
        break
    elif start > guess_me:
        print('oops')
        break
    start += 1
```

Если вы сделали все правильно, то увидите следующие строки:

```
too low
too low
too low
too low
too low
too low
found it!
```

Обратите внимание на то, что строка `elif start > guess_me:` могла содержать обычный оператор `else:`, поскольку, если значение `start` не меньше и не равно значению `guess_me`, оно должно быть больше. По крайней мере в этой Вселенной.

3. Используйте цикл `for`, чтобы вывести на экран значения списка `[3, 2, 1, 0]`:

```
>>> for value in [3, 2, 1, 0]:
...     print(value)
...
3
2
1
0
```

4. Используйте включение списка, чтобы создать список, который содержит нечетные числа в диапазоне `range(10)`:

```
>>> even = [number for number in range(10) if number % 2 == 0]
>>> even
[0, 2, 4, 6, 8]
```

5. Используйте включение словаря, чтобы создать словарь `squares`. Используйте вызов `range(10)`, чтобы получить ключи, и возведите их в квадрат, чтобы получить их значения:

```
>>> squares = {key: key*key for key in range(10)}
>>> squares
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81}
```

6. Используйте включение множества, чтобы создать множество `odd`, которое содержит четные числа в диапазоне `range(10)`:

```
>>> odd = {number for number in range(10) if number % 2 == 1}
>>> odd
{1, 3, 9, 5, 7}
```

7. Используйте включение генератора, чтобы вернуть строку `'Got '` и количество чисел в диапазоне `range(10)`. Итерируйте по нему с помощью цикла `for`:

```
>>> for thing in ('Got %s' % number for number in range(10)):
...     print(thing)
```

```
...
Got 0
Got 1
Got 2
Got 3
Got 4
Got 5
Got 6
Got 7
Got 8
Got 9
```

8. Определите функцию good, которая возвращает список ['Harry', 'Ron', 'Hermione']:

```
>>> def good():
...     return ['Harry', 'Ron', 'Hermione']
...
>>> good()
['Harry', 'Ron', 'Hermione']
```

9. Определите функцию генератора get_odds, которая возвращает нечетные числа из диапазона range(10). Используйте цикл for, чтобы найти и вывести третье возвращенное значение:

```
>>> def get_odds():
...     for number in range(1, 10, 2):
...         yield number
...
>>> for count, number in enumerate(get_odds(), 1):
...     if count == 3:
...         print("The third odd number is", number)
...         break
...
```

Третье нечетное число равно 5.

10. Определите декоратор test, который выводит строку 'start', когда вызывается функция, и строку 'end', когда функция завершает свою работу:

```
>>> def test(func):
...     def new_func(*args, **kwargs):
...         print('start')
...         result = func(*args, **kwargs)
...         print('end')
...         return result
...     return new_func
...
>>>
>>> @test
... def greeting():
```

```
...     print("Greetings, Earthling")
...
>>> greeting()
start
Greetings, Earthling
end
```

11. Определите исключение, которое называется `OopsException`. Сгенерируйте его, чтобы увидеть, что произойдет. Затем напишите код, позволяющий поймать это исключение и вывести строку `'Caught an oops'`:

```
>>> class OopsException(Exception):
...     pass
...
>>> raise OopsException()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
__main__.OopsException
>>>
>>> try:
...     raise OopsException
... except OopsException:
...     print('Caught an oops')
...
Caught an oops
```

12. Используйте функцию `zip()`, чтобы создать словарь `movies`, который объединяет в пары эти списки: `titles = ['Creature of Habit', 'Crewel Fate']` и `plots = ['A nun turns into a monster', 'A haunted yarn shop']`:

```
>>> titles = ['Creature of Habit', 'Crewel Fate']
>>> plots = ['A nun turns into a monster', 'A haunted yarn shop']
>>> movies = dict(zip(titles, plots))
>>> movies
{'Crewel Fate': 'A haunted yarn shop', 'Creature of Habit': 'A nun turns into
a monster'}
```

Глава 5. Py Boxes: модули, пакеты и программы

1. Создайте файл, который называется `zoo.py`. В этом файле объявите функцию `hours()`, которая выводит на экран строку `'Open 9-5 daily'`. Далее используйте интерактивный интерпретатор, чтобы импортировать модуль zoo и вызвать его функцию `hours()`.

   Так выглядит файл `zoo.py`:

```
def hours():
    print('Open 9-5 daily')
```

А теперь импортируем его интерактивно:

```
>>> import zoo
>>> zoo.hours()
Open 9-5 daily
```

2. В интерактивном интерпретаторе импортируйте модуль zoo под именем menagerie и вызовите его функцию `hours()`:

```
>>> import zoo as menagerie
>>> menagerie.hours()
Open 9-5 daily
```

3. Оставаясь в интерпретаторе, импортируйте непосредственно функцию `hours()` из модуля zoo и вызовите ее.

```
>>> from zoo import hours
>>> hours()
Open 9-5 daily
```

4. Импортируйте функцию `hours()` под именем `info` и вызовите ее:

```
>>> from zoo import hours as info
>>> info()
Open 9-5 daily
```

5. Создайте словарь с именем `plain`, содержащий пары «ключ — значение» `'a': 1`, `'b': 2` и `'c':3`, а затем выведите его на экран:

```
>>> plain = {'a': 1, 'b': 2, 'c': 3}
>>> plain
{'a': 1, 'c': 3, 'b': 2}
```

6. Создайте `OrderedDict` с именем `fancy` из пар «ключ — значение», приведенных в упражнении 5, и выведите его на экран. Изменился ли порядок ключей?

```
>>> from collections import OrderedDict
>>> fancy = OrderedDict([('a', 1), ('b', 2), ('c', 3)])
>>> fancy
OrderedDict([('a', 1), ('b', 2), ('c', 3)])
```

7. Создайте `defaultdict` с именем `dict_of_lists` и передайте ему аргумент `list`. Создайте список `dict_of_lists['a']` и присоедините к нему значение `'something for a'` за одну операцию. Выведите на экран `dict_of_lists['a']`:

```
>>> from collections import defaultdict
>>> dict_of_lists = defaultdict(list)
>>> dict_of_lists['a'].append('something for a')
>>> dict_of_lists['a']
['something for a']
```

Глава 6. Ой-ой-ой: объекты и классы

1. Создайте класс, который называется `Thing`, не имеющий содержимого, и выведите его на экран. Затем создайте объект `example` этого класса и также выведите его. Совпадают ли выведенные значения?

```
>>> class Thing:
...     pass
...
>>> print(Thing)
<class '__main__.Thing'>
>>> example = Thing()
>>> print(example)
<__main__.Thing object at 0x1006f3fd0>
```

2. Создайте новый класс с именем `Thing2` и присвойте его атрибуту `letters` значение `'abc'`. Выведите на экран значение атрибута `letters`:

```
>>> class Thing2:
...     letters = 'abc'
...
>>> print(Thing2.letters)
abc
```

3. Создайте еще один класс, который, конечно же, называется `Thing3`. В этот раз присвойте значение `'xyz'` атрибуту объекта, который называется `letters`. Выведите на экран значение атрибута `letters`. Понадобилось ли вам создавать объект класса, чтобы сделать это?

```
>>> class Thing3:
...     def __init__(self):
...         self.letters = 'xyz'
...
```

Переменная `letters` принадлежит любому объекту класса `Thing3`, но не самому классу `Thing3`:

```
>>> print(Thing3.letters)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: type object 'Thing3' has no attribute 'letters'
>>> something = Thing3()
>>> print(something.letters)
xyz
```

4. Создайте класс, который называется `Element`, имеющий атрибуты объекта `name`, `symbol` и `number`. Создайте объект этого класса со значениями `'Hydrogen'`, `'H'` и `1`:

```
>>> class Element:
...     def __init__(self, name, symbol, number):
```

```
...         self.name = name
...         self.symbol = symbol
...         self.number = number
...
>>> hydrogen = Element('Hydrogen', 'H', 1)
```

5. Создайте словарь со следующими ключами и значениями: `'name': 'Hydrogen'`, `'symbol': 'H'`, `'number': 1`. Далее создайте объект с именем `hydrogen` класса `Element` с помощью этого словаря.

   Начнем со словаря:

```
>>> el_dict = {'name': 'Hydrogen', 'symbol': 'H', 'number': 1}
```

   Это работает, однако необходимо напечатать много текста:

```
>>> hydrogen = Element(el_dict['name'], el_dict['symbol'], el_dict['number'])
```

   Убедимся, что это работает:

```
>>> hydrogen.name
'Hydrogen'
```

   Однако вы также можете инициализировать объект непосредственно с помощью словаря, поскольку его ключ `names` совпадает с аргументами функции `__init__` (аргументы — ключевые слова рассматриваются в главе 3):

```
>>> hydrogen = Element(**el_dict)
>>> hydrogen.name
'Hydrogen'
```

6. Для класса `Element` определите метод с именем `dump()`, который выводит на экран значения атрибутов объекта (`name`, `symbol` и `number`). Создайте объект `hydrogen` из этого нового определения и используйте метод `dump()`, чтобы вывести на экран его атрибуты:

```
>>> class Element:
...     def __init__(self, name, symbol, number):
...         self.name = name
...         self.symbol = symbol
...         self.number = number
...     def dump(self):
...         print('name=%s, symbol=%s, number=%s' %
...             (self.name, self.symbol, self.number))
...
>>> hydrogen = Element(**el_dict)
>>> hydrogen.dump()
name=Hydrogen, symbol=H, number=1
```

7. Вызовите функцию `print(hydrogen)`. В определении класса `Element` измените имя метода `dump` на `__str__`, создайте новый объект `hydrogen` и затем снова вызовите метод `print(hydrogen)`:

```
>>> print(hydrogen)
<__main__.Element object at 0x1006f5310>
>>> class Element:
...     def __init__(self, name, symbol, number):
...         self.name = name
...         self.symbol = symbol
...         self.number = number
...     def __str__(self):
...         return ('name=%s, symbol=%s, number=%s' %
...             (self.name, self.symbol, self.number))
...
>>> hydrogen = Element(**el_dict)
>>> print(hydrogen)
name=Hydrogen, symbol=H, number=1
```

`__str__()` — это один из волшебных методов Python. Функция `print` вызывает метод объекта `__str__()`, чтобы получить его строковое представление. Если у объекта нет метода `__str__()`, он получает метод по умолчанию от его родительского класса `Object`, который возвращает строку наподобие `<__main__.Element object at 0x1006f5310>`.

8. Модифицируйте класс `Element`, сделав атрибуты `name`, `symbol` и `number` закрытыми. Определите геттер, возвращающий значение атрибута, для каждого из них:

```
>>> class Element:
...     def __init__(self, name, symbol, number):
...         self.__name = name
...         self.__symbol = symbol
...         self.__number = number
...     @property
...     def name(self):
...         return self.__name
...     @property
...     def symbol(self):
...         return self.__symbol
...     @property
...     def number(self):
...         return self.__number
...
>>> hydrogen = Element('Hydrogen', 'H', 1)
```

```
>>> hydrogen.name
'Hydrogen'
>>> hydrogen.symbol
'H'
>>> hydrogen.number
1
```

9. Определите три класса: `Bear`, `Rabbit` и `Octothorpe`. Для каждого из них определите всего один метод — `eats()`. Этот метод должен возвращать значения `'berries'` (для `Bear`), `'clover'` (для `Rabbit`) или `'campers'` (для `Octothorpe`). Создайте по одному объекту каждого класса и выведите на экран то, что он ест:

```
>> class Bear:
...     def eats(self):
...         return 'berries'
...
>>> class Rabbit:
...     def eats(self):
...         return 'clover'
...
>>> class Octothorpe:
...     def eats(self):
...         return 'campers'
...
>>> b = Bear()
>>> r = Rabbit()
>>> o = Octothorpe()
>>> print(b.eats())
berries
>>> print(r.eats())
clover
>>> print(o.eats())
campers
```

10. Определите три класса: `Laser`, `Claw` и `SmartPhone`. Каждый из них имеет только один метод — `does()`. Он возвращает значения `'disintegrate'` (для `Laser`), `'crush'` (для `Claw`) или `'ring'` (для `SmartPhone`). Далее определите класс `Robot`, который содержит по одному объекту каждого из этих классов. Определите метод `does()` для класса `Robot`, который выводит на экран все, что делают его компоненты:

```
>>> class Laser:
...     def does(self):
...         return 'disintegrate'
...
>>> class Claw:
...     def does(self):
...         return 'crush'
...
```

```
>>> class SmartPhone:
...     def does(self):
...         return 'ring'
...
>>> class Robot:
...     def __init__(self):
...         self.laser = Laser()
...         self.claw = Claw()
...         self.smartphone = SmartPhone()
...     def does(self):
...         return '''I have many attachments:
... My laser, to %s.
... My claw, to %s.
... My smartphone, to %s.''' % (
...     self.laser.does(),
...     self.claw.does(),
...     self.smartphone.does() )
...
>>> robbie = Robot()
>>> print(robbie.does())
I have many attachments:
My laser, to disintegrate.
My claw, to crush.
My smartphone, to ring.
```

Глава 7. Работаем с данными профессионально

1. Создайте строку Unicode с именем mystery и присвойте ей значение '\U0001f4a9'. Выведите на экран значение строки mystery. Найдите имя Unicode для mystery:

```
>>> import unicodedata
>>> mystery = '\U0001f4a9'
>>> mystery
' '
>>> unicodedata.name(mystery)
'PILE OF POO'
```

Ой-ой-ой! Что еще у них там есть?

2. Закодируйте строку mystery, в этот раз с использованием кодировки UTF-8, в переменную типа bytes с именем pop_bytes. Выведите на экран значение переменной pop_bytes:

```
>>> pop_bytes = mystery.encode('utf-8')
>>> pop_bytes
b'\xf0\x9f\x92\xa9'
```

3. Используя кодировку UTF-8, декодируйте переменную pop_bytes в строку pop_string. Выведите на экран значение переменной pop_string. Равно ли оно значению переменной mystery?

```
>>> pop_string = pop_bytes.decode('utf-8')
>>> pop_string
' '
>>> pop_string == mystery
True
```

4. Запишите следующее стихотворение с помощью старого стиля форматирования. Подставьте строки 'roast beef', 'ham', 'head' и 'clam' в эту строку:

```
My kitty cat likes %s,
My kitty cat likes %s,
My kitty cat fell on his %s
And now thinks he's a %s.
>>> poem = '''
... My kitty cat likes %s,
... My kitty cat likes %s,
... My kitty cat fell on his %s
... And now thinks he's a %s.
... '''
>>> args = ('roast beef', 'ham', 'head', 'clam')
>>> print(poem % args)
My kitty cat likes roast beef,
My kitty cat likes ham,
My kitty cat fell on his head
And now thinks he's a clam.
```

5. Запишите следующее письмо по форме с помощью форматирования нового стиля. Сохраните строку под именем letter (это имя вы используете в следующем упражнении):

```
Dear {salutation} {name},
Thank you for your letter. We are sorry that our {product} {verbed} in your
{room}. Please note that it should never be used in a {room}, especially
near any {animals}.
Send us your receipt and {amount} for shipping and handling. We will send
you another {product} that, in our tests, is {percent}% less likely to
have {verbed}.
Thank you for your support.
Sincerely,
{spokesman}
{job_title}
>>> letter = '''
... Dear {salutation} {name},
```

```
...
... Thank you for your letter. We are sorry that our {product} {verb} in your
... {room}. Please note that it should never be used in a {room}, especially
... near any {animals}.
...
... Send us your receipt and {amount} for shipping and handling. We will send
... you another {product} that, in our tests, is {percent}% less likely to
... have {verbed}.
...
... Thank you for your support.
...
... Sincerely,
... {spokesman}
... {job_title}
... '''
```

6. Создайте словарь с именем `response`, имеющий значения для строковых ключей `'salutation'`, `'name'`, `'product'`, `'verbed'` (прошедшее время от слова глагола verb), `'room'`, `'animals'`, `'amount'`, `'percent'`, `'spokesman'` и `'job_title'`. Выведите на экран значение переменной `letter`, в которую подставлены значения из словаря `response`:

```
>>> response = {
...     'salutation': 'Colonel',
...     'name': 'Hackenbush',
...     'product': 'duck blind',
...     'verbed': 'imploded',
...     'room': 'conservatory',
...     'animals': 'emus',
...     'amount': '$1.38',
...     'percent': '1',
...     'spokesman': 'Edgar Schmeltz',
...     'job_title': 'Licensed Podiatrist'
...     }
...
>>> print(letter.format(**response))
Dear Colonel Hackenbush,
Thank you for your letter. We are sorry that our duck blind imploded in your
conservatory. Please note that it should never be used in a conservatory,
especially near any emus.
Send us your receipt and $1.38 for shipping and handling. We will send
you another duck blind that, in our tests, is 1% less likely to have imploded.
Thank you for your support.
Sincerely,
Edgar Schmeltz
Licensed Podiatrist
```

7. При работе с текстом вам могут пригодиться регулярные выражения. Мы воспользуемся ими несколькими способами в следующем примере текста. Перед вами стихотворение *Ode on the Mammoth Cheese*, написанное Джеймсом Макинтайром (James McIntyre) в 1866 году во славу головки сыра весом 7000 фунтов, которая была сделана в Онтарио и отправлена в международное путешествие. Если не хотите вводить это стихотворение целиком, используйте свой любимый поисковик и скопируйте его текст в программу. Или скопируйте его из проекта «Гутенберг». Назовите следующую строку `mammoth`:

```
>>> mammoth = '''
We have seen thee, queen of cheese,
Lying quietly at your ease,
Gently fanned by evening breeze,
Thy fair form no flies dare seize.
All gaily dressed soon you'll go
To the great Provincial show,
To be admired by many a beau
In the city of Toronto.
Cows numerous as a swarm of bees,
Or as the leaves upon the trees,
It did require to make thee please,
And stand unrivalled, queen of cheese.
May you not receive a scar as
We have heard that Mr. Harris
Intends to send you off as far as
The great world's show at Paris.
Of the youth beware of these,
For some of them might rudely squeeze
And bite your cheek, then songs or glees
We could not sing, oh! queen of cheese.
We'rt thou suspended from balloon,
You'd cast a shade even at noon,
Folks would think it was the moon
About to fall and crush them soon.
... '''
```

8. Импортируйте модуль `re`, чтобы использовать функции регулярных выражений в Python. Используйте функцию `re.findall()`, чтобы вывести на экран все слова, которые начинаются с буквы «c».

   Мы определим переменную `pat` для шаблона и затем будем искать такой шаблон в строке `mammoth`:

```
>>> import re
>>> re = r'\bc\w*'
```

```
>>> re.findall(pat, mammoth)
['cheese', 'city', 'cheese', 'cheek', 'could', 'cheese', 'cast', 'crush']
```

\b означает, что нужно начать с границы между словом и не словом. Используйте такую конструкцию, чтобы указать либо на начало, либо на конец слова. Литерал c — это первая буква всех слов, которые мы ищем. \w означает любой символ слова, которое включает в себя буквы, цифры и подчеркивания (_). * означает ноль или больше таких символов. Целиком это выражение находит слова, которые начинаются с «c», включая слово c. Если вы не использовали простую строку (у таких строк r стоит прямо перед открывающей кавычкой), Python интерпретирует \b как возврат на шаг и поиск по таинственной причине ничего не найдет:

```
>>> pat = '\bc\w*'
>>> re.findall(pat, mammoth)
[]
```

9. Найдите все четырехбуквенные слова, которые начинаются с буквы «c»:

```
>>> pat = r'\bc\w{3}\b'
>>> re.findall(pat, mammoth)
['city', 'cast']
```

Вам нужен последний символ \b, чтобы указать на конец слова. В противном случае вы получите первые четыре буквы всех слов, которые начинаются с «c» и имеют как минимум четыре буквы:

```
>>> pat = r'\bc\w{3}'
>>> re.findall(pat, mammoth)
['chee', 'city', 'chee', 'chee', 'coul', 'chee', 'cast', 'crus']
```

10. Найдите все слова, которые заканчиваются на букву «r».

Это упражнение с подвохом. Мы получаем правильный результат для слов, которые заканчиваются на «r»:

```
>>> pat = r'\b\w*r\b'
>>> re.findall(pat,mammoth)
['your', 'fair', 'Or', 'scar', 'Mr', 'far', 'For', 'your', 'or']
```

Однако результаты будут не так хороши, если мы поищем слова, которые заканчиваются на «l»:

```
>>> pat = r'\b\w*l\b'
>>> re.findall(pat,mammoth)
['All', 'll', 'Provincial', 'fall']
```

Что здесь делает буквосочетание «ll»? Паттерн \w совпадает только с буквами, цифрами и подчеркиваниями, но не с апострофами ASCII. В результате вы увидите буквосочетание «ll». Мы можем обработать этот крайний случай,

добавив апостроф в список символов, с которыми должен совпасть набор символов. Наша первая попытка не работает:

```
>>> >>> pat = r'\b[\w']*l\b'
  File "<stdin>", line 1
    pat = r'\b[\w']*l\b'
```

Python указывает на окрестности ошибки, но может потребоваться какое-то время, чтобы увидеть: ошибка заключалась в том, что строка шаблона окружена такими же апострофами — символами кавычки. Один из способов решить эту проблему — использовать управляющую последовательность с обратным слешем:

```
>>> pat = r'\b[\w\']*l\b'
>>> re.findall(pat, mammoth)
['All', "you'll", 'Provincial', 'fall']
```

Еще одно решение — окружить строку шаблона двойными кавычками:

```
>>> pat = r"\b[\w']*l\b"
>>> re.findall(pat, mammoth)
['All', "you'll", 'Provincial', 'fall']
```

11. Найдите все слова, которые содержат три гласные подряд.

Начиная с границы слова, любое число символов слова, три гласные и далее любые символы, не являющиеся гласными, до конца слова:

```
>>> pat = r'\b\w*[aeiou]{3}[^aeiou]\w*\b'
>>> re.findall(pat, mammoth)
['queen', 'quietly', 'beau\nIn', 'queen', 'squeeze', 'queen']
```

Выглядит правильно, за исключением строки `'beau\nIn'`. Мы искали строку `mammoth` целиком. Конструкция `[^aeiou]` совпадает с любыми символами, не являющимися гласными, включая `\n` (перенос строки, который отмечает конец текстовой строки). Нам нужно добавить еще кое-что в набор игнорируемых символов: `\s` совпадает с любыми символами пробелов, включая `\n`:

```
>>> pat = r'\b\w*[aeiou]{3}[^aeiou\s]\w*\b'
>>> re.findall(pat, mammoth)
['queen', 'quietly', 'queen', 'squeeze', 'queen']
```

В этот раз мы не нашли слово `beau`, поэтому нужно внести в шаблон еще одно исправление: совпадение с любым числом (даже нулем) не гласных после трех гласных. Наш предыдущий шаблон всегда совпадал с одним не гласным символом:

```
>>> pat = r'\b\w*[aeiou]{3}[^aeiou\s]*\w*\b'
>>> re.findall(pat, mammoth)
['queen', 'quietly', 'beau', 'queen', 'squeeze', 'queen']
```

Что это показывает? Среди всего прочего, что регулярные выражения могут сделать многое, но их может быть очень трудно написать правильно.

12. Используйте метод `unhexlify` для того, чтобы преобразовать эту шестнадцатеричную строку (созданную путем объединения двух строк, что позволило ей разместиться на странице) в переменную типа `bytes` с именем `gif`:

```
'47494638396101000100800000000000ffffff21f9' +
'0401000000002c000000000100010000020144003b'
>>> import binascii
>>> hex_str = '47494638396101000100800000000000ffffff21f9' + \
...     '0401000000002c000000000100010000020144003b'
>>> gif = binascii.unhexlify(hex_str)
>>> len(gif)
42
```

13. Байты, содержащиеся в переменной `gif`, определяют однопиксельный прозрачный GIF-файл. Этот формат является одним из самых распространенных. Корректный файл формата GIF начинается со строки `GIF89a`. Является ли этот файл корректным?

```
>>> gif[:6] == b'GIF89a'
True
```

Обратите внимание на то, что нам нужно использовать `b`, чтобы указать, что строка состоит из байтов, а не из символов Unicode. Вы можете сравнить байты с байтами, но не можете сравнивать байты и символы:

```
>>> gif[:6] == 'GIF89a'
False
>>> type(gif)
<class 'bytes'>
>>> type('GIF89a')
<class 'str'>
>>> type(b'GIF89a')
<class 'bytes'>
```

14. Ширина файла формата GIF является шестнадцатибитным целым числом с обратным порядком байтов, которое начинается со смещения 6 байт. Его высота имеет такой же размер и начинается со смещения 8 байт. Извлеките и выведите на экран эти значения для переменной `gif`. Равны ли они 1?

```
>>> import struct
>>> width, height = struct.unpack('<HH', gif[6:10])
>>> width, height
(1, 1)
```

Глава 8. Данные должны куда-то попадать

1. Присвойте строку `'This is a test of the emergency text system'` переменной `test1` и запишите переменную `test1` в файл с именем `test.txt`:

```
>>> test1 = 'This is a test of the emergency text system'
>>> len(test1)
43
```

Вот как можно сделать это с помощью функций `open`, `write` и `close`:

```
>>> outfile = open('test.txt', 'wt')
>>> outfile.write(test1)
43
>>> outfile.close()
```

Или можете использовать `with` и избежать вызова `close` (Python сделает это за вас):

```
>>> with open('test.txt', 'wt') as outfile:
...     outfile.write(test1)
...
43
```

2. Откройте файл `test.txt` и считайте его содержимое в строку `test2`. Совпадают ли строки `test1` и `test2`?

```
>>> with open('test.txt', 'rt') as infile:
...     test2 = infile.read()
...
>>> len(test2)
43
>>> test1 == test2
True
```

3. Сохраните следующие несколько строк в файл `books.csv`. Обратите внимание на то, что, если поля разделены запятыми, вам нужно заключить поле в кавычки, если оно содержит запятую:

```
author,book
J R R Tolkien,The Hobbit
Lynne Truss,"Eats, Shoots & Leaves"
>>> text = '''author,book
... J R R Tolkien,The Hobbit
... Lynne Truss,"Eats, Shoots & Leaves"
... '''
>>> with open('test.csv', 'wt') as outfile:
...     outfile.write(text)
...
73
```

4. Используйте модуль csv и его метод DictReader, чтобы считать содержимое файла books.csv в переменную books. Выведите на экран значения переменной books. Обработал ли метод DictReader кавычки и запятые в заголовке второй книги?

```
>>> with open('test.csv', 'rt') as infile:
...     books = csv.DictReader(infile)
...     for book in books:
...         print(book)
...
{'book': 'The Hobbit', 'author': 'J R R Tolkien'}
{'book': 'Eats, Shoots & Leaves', 'author': 'Lynne Truss'}
```

5. Создайте CSV-файл с именем books.csv и запишите его в следующие строки:

```
title,author,year
The Weirdstone of Brisingamen,Alan Garner,1960
Perdido Street Station,China Miéville,2000
Thud!,Terry Pratchett,2005
The Spellman Files,Lisa Lutz,2007
Small Gods,Terry Pratchett,1992
>>> text = '''title,author,year
... The Weirdstone of Brisingamen,Alan Garner,1960
... Perdido Street Station,China Miéville,2000
... Thud!,Terry Pratchett,2005
... The Spellman Files,Lisa Lutz,2007
... Small Gods,Terry Pratchett,1992
... '''
>>> with open('books.csv', 'wt') as outfile:
...     outfile.write(text)
...
201
```

6. Используйте модуль sqlite3, чтобы создать базу данных SQLite books.db и таблицу books, содержащую следующие поля: title (text), author (text) и year (integer):

```
>>> import sqlite3
>>> db = sqlite3.connect('books.db')
>>> curs = db.cursor()
>>> curs.execute('''create table book (title text, author text, year int)''')
<sqlite3.Cursor object at 0x1006e3b90>
>>> db.commit()
```

7. Считайте данные из файла books.csv и добавьте их в таблицу book:

```
>>> import csv
>>> import sqlite3
>>> ins_str = 'insert into book values(?, ?, ?)'
>>> with open('books.csv', 'rt') as infile:
...     books = csv.DictReader(infile)
...     for book in books:
...        curs.execute(ins_str, (book['title'], book['author'], book['year']))
```

```
...
<sqlite3.Cursor object at 0x1007b21f0>
<sqlite3.Cursor object at 0x1007b21f0>
<sqlite3.Cursor object at 0x1007b21f0>
<sqlite3.Cursor object at 0x1007b21f0>
<sqlite3.Cursor object at 0x1007b21f0>
>>> db.commit()
```

8. Считайте и выведите на экран графу `title` таблицы `book` в алфавитном порядке:

```
>>> sql = 'select title from book order by title asc'
>>> for row in db.execute(sql):
...     print(row)
...
('Perdido Street Station',)
('Small Gods',)
('The Spellman Files',)
('The Weirdstone of Brisingamen',)
('Thud!',)
```

Если вы хотите вывести на экран значение `title`, не пользуясь конструкциями для работы с кортежем (круглыми скобками и запятой), попробуйте следующее:

```
>>> for row in db.execute(sql):
...     print(row[0])
...
Perdido Street Station
Small Gods
The Spellman Files
The Weirdstone of Brisingamen
Thud!
```

Если хотите проигнорировать начальное слово `'The'` в заголовках, вам нужно написать еще одну строку SQL:

```
>>> sql = '''select title from book order by
... case when (title like "The %") then substr(title, 5) else title end'''
>>> for row in db.execute(sql):
...     print(row[0])
...
Perdido Street Station
Small Gods
The Spellman Files
Thud!
The Weirdstone of Brisingamen
```

9. Считайте и выведите на экран все графы таблицы `book` в порядке публикации:

```
>>> for row in db.execute('select * from book order by year'):
...     print(row)
```

```
...
('The Weirdstone of Brisingamen', 'Alan Garner', 1960)
('Small Gods', 'Terry Pratchett', 1992)
('Perdido Street Station', 'China Miéville', 2000)
('Thud!', 'Terry Pratchett', 2005)
('The Spellman Files', 'Lisa Lutz', 2007)
```

Чтобы вывести на экран все поля каждого ряда, просто разделите их запятой и пробелом:

```
>>> for row in db.execute('select * from book order by year'):
...     print(*row, sep=', ')
...
The Weirdstone of Brisingamen, Alan Garner, 1960
Small Gods, Terry Pratchett, 1992
Perdido Street Station, China Miéville, 2000
Thud!, Terry Pratchett, 2005
The Spellman Files, Lisa Lutz, 2007
```

10. Используйте модуль `sqlalchemy`, чтобы подключиться к базе данных `sqlite3` `books.db`, которую вы только что создали в упражнении 6. Как и в упражнении 8, считайте и выведите на экран графу `title` таблицы `book` в алфавитном порядке:

```
>>> import sqlalchemy
>>> conn = sqlalchemy.create_engine('sqlite:///books.db')
>>> sql = 'select title from book order by title asc'
>>> rows = conn.execute(sql)
>>> for row in rows:
...     print(row)
...
('Perdido Street Station',)
('Small Gods',)
('The Spellman Files',)
('The Weirdstone of Brisingamen',)
('Thud!',)
```

11. Установите сервер Redis и библиотеку Python Redis (с помощью команды `pip install redis`) на свой компьютер. Создайте хеш Redis с именем `test`, содержащий поля `count (1)` и `name ('Fester Bestertester')`. Выведите все поля хеша `test`:

```
>>> import redis
>>> conn = redis.Redis()
>>> conn.delete('test')
1
>>> conn.hmset('test', {'count': 1, 'name': 'Fester Bestertester'})
True
>>> conn.hgetall('test')
{b'name': b'Fester Bestertester', b'count': b'1'}
```

12. Увеличьте поле count хеша test и выведите его на экран:

```
>>> conn.hincrby('test', 'count', 3)
4
>>> conn.hget('test', 'count')
b'4'
```

Глава 9. Распутываем Всемирную паутину

1. Если вы еще не установили Flask, сделайте это сейчас. Это также установит werkzeug, jinja2 и, возможно, другие пакеты.
2. Создайте скелет сайта с помощью веб-сервера Flask. Убедитесь, что сервер начинает свою работу по адресу Localhost на стандартном порте 5000. Если ваш компьютер уже использует порт 5000 для чего-то еще, воспользуйтесь другим портом.

   Так выглядит файл flask1.py:

```
from flask import Flask
app = Flask(__name__)
app.run(port=5000, debug=True)
```

   Поехали:

```
$ python flask1.py
 * Running on http://127.0.0.1:5000/
 * Restarting with reloader
```

3. Добавьте функцию home(), которая обрабатывает запросы к домашней странице. Укажите ей возвращать строку запроса It's alive!.

   Как нам назвать этот файл, flask2.py?

```
from flask import Flask
app = Flask(__name__)
@app.route('/')
def home():
    return "It's alive!"
app.run(debug=True)
```

   Запустим сервер:

```
$ python flask2.py
 * Running on http://127.0.0.1:5000/
 * Restarting with reloader
```

   Наконец, получим доступ к домашней странице через браузер, HTTP-программы командной строки вроде curl, или wget, или даже telnet:

```
 $ curl http://localhost:5000/
It's alive!
```

4. Создайте шаблон для jinja2, который называется `home.html` и содержит следующий контент:

```
I'm of course referring to {{thing}}, which is {{height}} feet tall and {{color}}.
```

Создайте папку **templates** и файл `home.html`, содержащий показанное. Если ваш сервер Flask все еще работает после запуска предыдущих примеров, он обнаружит новый контент и перезапустится.

5. Модифицируйте функцию `home()` вашего сервера, чтобы она использовала шаблон `home.html`. Передайте ей три параметра для команды `GET`: `thing`, `height` и `color`.

Перед вами файл `flask3.py`:

```
from flask import Flask, request, render_template
app = Flask(__name__)
@app.route('/')
def home():
    thing = request.values.get('thing')
    height = request.values.get('height')
    color = request.values.get('color')
    return render_template('home.html',
        thing=thing, height=height, color=color)
app.run(debug=True)
```

Перейдите в своем клиенте по следующему адресу:

```
http://localhost:5000/?thing=Octothorpe&height=7&color=green
```

Вы должны увидеть следующее:

```
I'm of course referring to Octothorpe, which is 7 feet tall and green.
```

Глава 10. Системы

1. Запишите текущие дату и время как строку в текстовый файл `today.txt`:

```
>>> from datetime import date
>>> now = date.today()
>>> now_str = now.isoformat()
>>> with open('today', 'wt') as output:
...     print(now_str, file=output)
>>>
```

Вместо функции `print` вы могли бы использовать строку вроде `output.write(now_str)`. Использование функции `print` добавляет символ перевода строки в конце.

2. Прочтите текстовый файл `today.txt` и разместите данные в строке `today_string`:

```
>>> with open('today', 'rt') as input:
...     today_string = input.read()
...
>>> today_string
'2014-02-04\n'
```

3. Разберите дату из строки today_string:

```
>>> fmt = '%Y-%m-%d\n'
>>> datetime.strptime(today_string, fmt)
datetime.datetime(2014, 2, 4, 0, 0)
```

Если вы записали тот символ новой строки в файл, вам нужно, чтобы он совпал со строкой формата.

4. Выведите на экран список файлов текущей папки.

Если ваша текущая папка называется ohmy и содержит три файла с именами по названиям животных, код может выглядеть так:

```
>>> import os
>>> os.listdir('.')
['bears', 'lions', 'tigers']
```

5. Выведите на экран список файлов родительской папки.

Если родительская папка содержит два файла и текущую папку ohmy, код может выглядеть так:

```
>>> import os
>>> os.listdir('..')
['ohmy', 'paws', 'whiskers']
```

6. Используйте модуль multiprocessing, чтобы создать три отдельных процесса. Заставьте каждый из них ждать случайное количество секунд (от одной до пяти), вывести текущее время и завершить работу.

Сохраните этот файл под именем multi_times.py:

```
import multiprocessing
def now(seconds):
    from datetime import datetime
    from time import sleep
    sleep(seconds)
    print('wait', seconds, 'seconds, time is', datetime.utcnow())
if __name__ == '__main__':
    import random
    for n in range(3):
        seconds = random.random()
        proc = multiprocessing.Process(target=now, args=(seconds,))
        proc.start()
$ python multi_times.py
wait 0.4670532005508353 seconds, time is 2014-06-03 05:14:22.930541
wait 0.5908421960431798 seconds, time is 2014-06-03 05:14:23.054925
wait 0.8127669040699719 seconds, time is 2014-06-03 05:14:23.275767
```

7. Создайте объект date, содержащий дату вашего рождения.

   Предположим, вы родились 14 августа 1982 года:

```
>>> my_day = date(1982, 8, 14)
>>> my_day
datetime.date(1982, 8, 14)
```

8. В какой день недели вы родились?

```
>>> my_day.weekday()
5
>>> my_day.isoweekday()
6
```

   Для weekday() значение для понедельника равно 0, а для воскресенья — 6. Для функции isoweekday() значение для понедельника равно 1, а для воскресенья — 7. Поэтому искомый день — суббота.

9. Когда вам будет (или уже было) 10 000 дней от роду?

```
>>> from datetime import timedelta
>>> party_day = my_day + timedelta(days=10000)
>>> party_day
datetime.date(2009, 12, 30)
```

   Если это был ваш день рождения, вы, возможно, пропустили еще один повод повеселиться.

Глава 11. Конкуренция и сети

1. Используйте объект класса socket, чтобы реализовать службу, сообщающую текущее время. Когда клиент отправляет на сервер строку 'time', верните текущие дату и время как строку ISO.

   Вот так можно написать сервер udp_time_server.py:

```
from datetime import datetime
import socket
address = ('localhost', 6789)
max_size = 4096
print('Starting the server at', datetime.now())
print('Waiting for a client to call.')
server = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
server.bind(address)
while True:
    data, client_addr = server.recvfrom(max_size)
    if data == b'time':
```

```
        now = str(datetime.utcnow())
        data = now.encode('utf-8')
        server.sendto(data, client_addr)
        print('Server sent', data)
server.close()
```

А так — клиент udp_time_client.py:

```
import socket
from datetime import datetime
from time import sleep
address    = ('localhost', 6789)
max_size   = 4096
print('Starting the client at', datetime.now())
client = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
while True:
    sleep(5)
    client.sendto(b'time', address)
    data, server_addr = client.recvfrom(max_size)
    print('Client read', data)
client.close()
```

Я поместил вызов sleep(5) в верхней части цикла клиента, чтобы сделать обмен данными менее быстрым. Запустите сервер в одном окне:

```
$ python udp_time_server.py
Starting the server at 2014-06-02 20:28:47.415176
Waiting for a client to call.
```

Запустите клиент в другом окне:

```
$ python udp_time_client.py
Starting the client at 2014-06-02 20:28:51.454805
```

Через 5 секунд вы начнете видеть сообщения в обоих окнах. Так выглядят первые три строки от сервера:

```
Server sent b'2014-06-03 01:28:56.462565'
Server sent b'2014-06-03 01:29:01.463906'
Server sent b'2014-06-03 01:29:06.465802'
```

А так — первые три строки от клиента:

```
Client read b'2014-06-03 01:28:56.462565'
Client read b'2014-06-03 01:29:01.463906'
Client read b'2014-06-03 01:29:06.465802'
```

Обе эти программы работают вечно, поэтому вам нужно завершать их вручную.

2. Используйте сокеты ZeroMQ REQ и REP, чтобы сделать то же самое.

   Так выглядит файл zmq_time_server.py:

```
import zmq
from datetime import datetime
```

```
host = '127.0.0.1'
port = 6789
context = zmq.Context()
server = context.socket(zmq.REP)
server.bind("tcp://%s:%s" % (host, port))
print('Server started at', datetime.utcnow())
while True:
    #  Wait for next request from client
    message = server.recv()
    if message == b'time':
        now = datetime.utcnow()
        reply = str(now)
        server.send(bytes(reply, 'utf-8'))
        print('Server sent', reply)
```

А так — zmq_time_client.py:

```
import zmq
from datetime import datetime
from time import sleep
host = '127.0.0.1'
port = 6789
context = zmq.Context()
client = context.socket(zmq.REQ)
client.connect("tcp://%s:%s" % (host, port))
print('Client started at', datetime.utcnow())
while True:
    sleep(5)
    request = b'time'
    client.send(request)
    reply = client.recv()
    print("Client received %s" % reply)
```

Для простых сокетов вам нужно сначала запустить сервер. С помощью ZeroMQ вы можете запустить первым как клиент, так и сервер:

```
$ python zmq_time_server.py
Server started at 2014-06-03 01:39:36.933532
$ python zmq_time_client.py
Client started at 2014-06-03 01:39:42.538245
```

Через 15 секунд вы должны увидеть сообщения от сервера:

```
Server sent 2014-06-03 01:39:47.539878
Server sent 2014-06-03 01:39:52.540659
Server sent 2014-06-03 01:39:57.541403
```

Эти строки вы должны увидеть в сообщении от клиента:

```
Client received b'2014-06-03 01:39:47.539878'
Client received b'2014-06-03 01:39:52.540659'
Client received b'2014-06-03 01:39:57.541403'
```

3. Попробуйте сделать то же самое с помощью `XMLRPC`.

   Сервер `xmlrpc_time_server.py`:

```
from xmlrpc.server import SimpleXMLRPCServer
def now():
    from datetime import datetime
    data = str(datetime.utcnow())
    print('Server sent', data)
    return data
server = SimpleXMLRPCServer(("localhost", 6789))
server.register_function(now, "now")
server.serve_forever()
```

   И клиент `xmlrpc_time_client.py`:

```
import xmlrpc.client
from time import sleep
proxy = xmlrpc.client.ServerProxy("http://localhost:6789/")
while True:
    sleep(5)
    data = proxy.now()
    print('Client received', data)
```

   Запустим сервер:

```
$ python xmlrpc_time_server.py
```

   Запустим клиент:

```
$ python xmlrpc_time_client.py
```

   Подождите примерно 15 секунд. Так выглядят первые три строки от сервера:

```
Server sent 2014-06-03 02:14:52.299122
127.0.0.1 -- [02/Jun/2014 21:14:52] "POST / HTTP/1.1" 200 -
Server sent 2014-06-03 02:14:57.304741
127.0.0.1 -- [02/Jun/2014 21:14:57] "POST / HTTP/1.1" 200 -
Server sent 2014-06-03 02:15:02.310377
127.0.0.1 -- [02/Jun/2014 21:15:02] "POST / HTTP/1.1" 200 -
```

   А так — первые три строки от клиента:

```
Client received 2014-06-03 02:14:52.299122
Client received 2014-06-03 02:14:57.304741
Client received 2014-06-03 02:15:02.310377
```

4. Возможно, вы видели эпизод телесериала I Love Lucy, в котором Люси и Этель работают на шоколадной фабрике (это классика). Парочка стала отставать, когда линия конвейера, которая направляла к ним на обработку конфеты, начала работать еще быстрее. Напишите симуляцию, которая отправляет разные типы конфет в список Redis, и клиент `Lucy`, который делает блокирующие вы-

талкивания из списка. Ей нужно 0,5 секунды, чтобы обработать одну конфету. Выведите на экран время и тип каждой конфеты, которую получит `Lucy`, а также количество необработанных конфет:

`redis_choc_supply.py` передает бесконечное количество конфет:

```
import redis
import random
from time import sleep
conn = redis.Redis()
varieties = ['truffle', 'cherry', 'caramel', 'nougat']
conveyor = 'chocolates'
while True:
    seconds = random.random()
    sleep(seconds)
    piece = random.choice(varieties)
    conn.rpush(conveyor, piece)
```

`redis_lucy.py` может выглядеть так:

```
import redis
from datetime import datetime
from time import sleep
conn = redis.Redis()
timeout = 10
conveyor = 'chocolates'
while True:
    sleep(0.5)
    msg = conn.blpop(conveyor, timeout)
    remaining = conn.llen(conveyor)
    if msg:
        piece = msg[1]
        print('Lucy got a', piece, 'at', datetime.utcnow(),
        ', only', remaining, 'left')
```

Запустите их в любом порядке. Поскольку Люси требуется полсекунды для обработки каждой конфеты и они появляются в среднем каждые полсекунды, это становится похоже на гонку. Чем раньше вы запустите конвейер, тем более сложной сделаете жизнь Люси:

```
$ python redis_choc_supply.py&
$ python redis_lucy.py
Lucy got a b'nougat' at 2014-06-03 03:15:08.721169 , only 4 left
Lucy got a b'cherry' at 2014-06-03 03:15:09.222816 , only 3 left
Lucy got a b'truffle' at 2014-06-03 03:15:09.723691 , only 5 left
Lucy got a b'truffle' at 2014-06-03 03:15:10.225008 , only 4 left
Lucy got a b'cherry' at 2014-06-03 03:15:10.727107 , only 4 left
Lucy got a b'cherry' at 2014-06-03 03:15:11.228226 , only 5 left
Lucy got a b'cherry' at 2014-06-03 03:15:11.729735 , only 4 left
Lucy got a b'truffle' at 2014-06-03 03:15:12.230894 , only 6 left
```

```
Lucy got a b'caramel' at 2014-06-03 03:15:12.732777 , only 7 left
Lucy got a b'cherry' at 2014-06-03 03:15:13.234785 , only 6 left
Lucy got a b'cherry' at 2014-06-03 03:15:13.736103 , only 7 left
Lucy got a b'caramel' at 2014-06-03 03:15:14.238152 , only 9 left
Lucy got a b'cherry' at 2014-06-03 03:15:14.739561 , only 8 left
```

Бедная Люси.

5. Используйте ZeroMQ, чтобы публиковать стихотворение из упражнения 7 главы 7 по одному слову за раз. Напишите потребителя ZeroMQ, который будет выводить на экран каждое слово, начинающееся с гласной. Напишите другого потребителя, который будет выводить все слова, состоящие из пяти букв. Знаки препинания игнорируйте.

   Так выглядит сервер, `poem_pub.py`, который отщипывает по одному слову стихотворения и публикует его в тему `vowels`, если оно начинается с гласной, и в тему `five`, если состоит из пяти букв. Некоторые слова могут оказаться в обеих темах, некоторые — ни в одной:

```
import string
import zmq
host = '127.0.0.1'
port = 6789
ctx = zmq.Context()
pub = ctx.socket(zmq.PUB)
pub.bind('tcp://%s:%s' % (host, port))
with open('mammoth.txt', 'rt') as poem:
    words = poem.read()
for word in words.split():
    word = word.strip(string.punctuation)
    data = word.encode('utf-8')
    if word.startswith(('a','e','i','o','u','A','e','i','o','u')):
        pub.send_multipart([b'vowels', data])
    if len(word) == 5:
        pub.send_multipart([b'five', data])
```

   Клиент `poem_sub.py` подписывается на темы `vowels` и `five` и выводит на экран тему и слово:

```
import string
import zmq
host = '127.0.0.1'
port = 6789
ctx = zmq.Context()
sub = ctx.socket(zmq.SUB)
sub.connect('tcp://%s:%s' % (host, port))
sub.setsockopt(zmq.SUBSCRIBE, b'vowels')
sub.setsockopt(zmq.SUBSCRIBE, b'five')
```

```
while True:
    topic, word = sub.recv_multipart()
    print(topic, word)
```

Если вы запустите эти программы, они не будут работать, хотя код выглядит хорошо. Вам нужно прочитать руководство ZeroMQ, чтобы узнать о проблеме медленного присоединившегося: даже если вы запустите клиент раньше сервера, сервер начнет отправлять данные сразу после запуска, а клиенту потребуется некоторое время, чтобы подключиться к серверу. Если вы публикуете сообщения постоянным потоком и не задумываетесь о том, когда к вам подключаются подписчики, это не проблема. Но в этом случае поток данных настолько короткий, что он заканчивается еще до того, как подписчик успеет моргнуть.

Простейший способ исправить это — заставить публикатора пропустить секунду после вызова метода `bind()` и до того, как он начнет отправлять сообщения. Назовем эту версию `poem_pub_sleep.py`:

```
import string
import zmq
from time import sleep
host = '127.0.0.1'
port = 6789
ctx = zmq.Context()
pub = ctx.socket(zmq.PUB)
pub.bind('tcp://%s:%s' % (host, port))
sleep(1)
with open('mammoth.txt', 'rt') as poem:
    words = poem.read()
for word in words.split():
    word = word.strip(string.punctuation)
    data = word.encode('utf-8')
    if word.startswith(('a','e','i','o','u','A','e','i','o','u')):
        print('vowels', data)
        pub.send_multipart([b'vowels', data])
    if len(word) == 5:
        print('five', data)
        pub.send_multipart([b'five', data])
```

Запустите подписчика, а затем и сонного публикатора:

```
$ python poem_sub.py
$ python poem_pub_sleep.py
```

Теперь у подписчика есть время на то, чтобы получить сообщения по выбранным темам. Так выглядят первые строки выходной информации:

```
b'five' b'queen'
b'vowels' b'of'
```

```
b'five' b'Lying'
b'vowels' b'at'
b'vowels' b'ease'
b'vowels' b'evening'
b'five' b'flies'
b'five' b'seize'
b'vowels' b'All'
b'five' b'gaily'
b'five' b'great'
b'vowels' b'admired'
```

Если вы не можете добавить вызов `sleep()` в код публикатора, вы можете синхронизировать публикатора и подписчика с помощью сокетов `REQ` и `REP`. Примеры файлов `publisher.py` и `subscriber.py` вы можете найти на GitHub.

E Вспомогательные материалы

Я обнаружил, что некоторые вещи мне приходится подсматривать слишком часто. Вот информация, которая, надеюсь, окажется вам полезной.

Приоритет операторов

Эта таблица — ремикс официальной документации о приоритетах для Python 3, операторы с самым высоким приоритетом находятся наверху.

Оператор	**Описание и примеры**
[v1, …], { v1, …}, { k1: v1, …}, (…)	Создание или включение списка/множества/словаря/генератора, выражение в скобках
seq [n], seq [n : m], func (args…), obj .attr	Индекс, разбиение, вызов функции, ссылка на атрибут
**	Экспонента
'+'x, '-'x, '~'x	Знаки «плюс» и «минус», битовое НЕ
*, /, //, %	Умножение, деление с плавающей точкой, целочисленное деление, напоминание
+, -	Сложение, вычитание
<<, >>	Битовый сдвиг вправо или влево
&	Битовое И
|	Битовое ИЛИ
in, not in, is, is not, <, <=, >, >=, !=, ==	Проверка на членство и равенство
not x	Булево (логическое) НЕ
and	Булево И
or	Булево ИЛИ
if … else	Условное выражение
lambda	Лямбда-выражение

Строковые методы

Python предлагает строковые методы (могут быть использованы с любым объектом str) и модуль string, содержащий полезные определения. Воспользуемся проверочными переменными:

```
>>> s = "OH, my paws and whiskers!"
>>> t = "I'm late!"
```

Изменение регистра

```
>>> s.capitalize()
'Oh, my paws and whiskers!'
>>> s.lower()
'oh, my paws and whiskers!'
>>> s.swapcase()
'oh, MY PAWS AND WHISKERS!'
>>> s.title()
'Oh, My Paws And Whiskers!'
>>> s.upper()
'OH, MY PAWS AND WHISKERS!'
```

Поиск

```
>>> s.count('w')
2
>>> s.find('w')
9
>>> s.index('w')
9
>>> s.rfind('w')
16
>>> s.rindex('w')
16
>>> s.startswith('OH')
True
```

Изменение

```
>>> ''.join(s)
'OH, my paws and whiskers!'
>>> ' '.join(s)
'O H ,   m y   p a w s   a n d   w h i s k e r s !'
>>> ' '.join((s, t))
"OH, my paws and whiskers! I'm late!"
```

```
>>> s.lstrip('HO')
', my paws and whiskers!'
>>> s.replace('H', 'MG')
'OMG, my paws and whiskers!'
>>> s.rsplit()
['OH,', 'my', 'paws', 'and', 'whiskers!']
>>> s.rsplit(' ', 1)
 ['OH, my paws and', 'whiskers!']
>>> s.split()
['OH,', 'my', 'paws', 'and', 'whiskers!']
>>> s.split(' ')
['OH,', 'my', 'paws', 'and', 'whiskers!']
>>> s.splitlines()
['OH, my paws and whiskers!']
>>> s.strip()
'OH, my paws and whiskers!'
>>> s.strip('s!')
'OH, my paws and whisker'
```

Форматирование

```
>>> s.center(30)
'  OH, my paws and whiskers!   '
>>> s.expandtabs()
'OH, my paws and whiskers!'
>>> s.ljust(30)
'OH, my paws and whiskers!     '
>>> s.rjust(30)
'     OH, my paws and whiskers!'
```

Тип строки

```
>>> s.isalnum()
False
>>> s.isalpha()
False
>>> s.isprintable()
True
>>> s.istitle()
False
>>> s.isupper()
False
>>> s.isdecimal()
False
>>> s.isnumeric()
False
```

Атрибуты модуля string

Существуют атрибуты класса, которые используются как определение констант.

<table>
<tr><th>Атрибут</th><th>Пример</th></tr>
<tr><td>ascii_letters</td><td>'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ '</td></tr>
<tr><td>ascii_lowercase</td><td>'abcdefghijklmnopqrstuvwxyz'</td></tr>
<tr><td>ascii_uppercase</td><td>'ABCDEFGHIJKLMNOPQRSTUVWXYZ'</td></tr>
<tr><td>digits</td><td>'0123456789'</td></tr>
<tr><td>hexdigits</td><td>'0123456789abcdefABCDEF'</td></tr>
<tr><td>octdigits</td><td>'01234567'</td></tr>
<tr><td>punctuation</td><td>"!"#$%&\'()*+,-./:;<=>?@[\]^_\{|}~'</td></tr>
<tr><td>printable</td><td>"0123456789abcdefghijklmnopqrstuvwxyz' + 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
+ '!"#$%&\'()*+,-./:;<=>?@[\]^_'{|}~' + ' \t\n\r\x0b\x0c'</td></tr>
<tr><td>whitespace</td><td>' \t\n\r\x0b\x0c'</td></tr>
</table>

Билл Любанович

Простой Python. Современный стиль программирования

Перевел с английского Е. Зазноба

Заведующий редакцией	*О. Сивченко*
Ведущий редактор	*Н. Гринчик*
Литературный редактор	*Н. Рощина*
Художник	*С. Заматевская*
Корректоры	*О. Андриевич, Е. Павлович*
Верстка	*Г. Блинов*

ООО «Питер Пресс», 192102, Санкт-Петербург, ул. Андреевская (д. Волкова), 3, литер А, пом. 7Н.

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

Подписано в печать 07.06.16. Формат 70×100/16. Бумага писчая. Усл. п. л. 38,700. Тираж 1200. Заказ 0000.

Отпечатано в ОАО «Первая Образцовая типография». Филиал «Чеховский Печатный Двор».
142300, Московская область, г. Чехов, ул. Полиграфистов, 1.
Сайт: www.chpk.ru. E-mail: marketing@chpk.ru
Факс: 8(496) 726-54-10, телефон: (495) 988-63-87

ИЗДАТЕЛЬСКИЙ ДОМ «ПИТЕР» предлагает профессиональную, популярную и детскую развивающую литературу

Заказать книги оптом можно в наших представительствах

РОССИЯ

Санкт-Петербург: м. «Выборгская», Б. Сампсониевский пр., д. 29а
тел./факс: (812) 703-73-83, 703-73-72; e-mail: sales@piter.com

Москва: м. «Электрозаводская», Семеновская наб., д. 2/1, стр. 1, 6 этаж
тел./факс: (495) 234-38-15; e-mail: sales@msk.piter.com

Воронеж: тел.: 8 951 861-72-70; e-mail: hitsenko@piter.com

Екатеринбург: ул. Толедова, д. 43а; тел./факс: (343) 378-98-41, 378-98-42;
e-mail: office@ekat.piter.com; skype: ekat.manager2

Нижний Новгород: тел.: 8 930 712-75-13; e-mail: yashny@yandex.ru; skype: yashny1

Ростов-на-Дону: ул. Ульяновская, д. 26
тел./факс: (863) 269-91-22, 269-91-30; e-mail: piter-ug@rostov.piter.com

Самара: ул. Молодогвардейская, д. 33а, офис 223
тел./факс: (846) 277-89-79, 277-89-66; e-mail: pitvolga@mail.ru,
pitvolga@samara-ttk.ru

БЕЛАРУСЬ

Минск: ул. Розы Люксембург, д. 163; тел./факс: +37 517 208-80-01, 208-81-25;
e-mail: og@minsk.piter.com

Издательский дом «Питер» приглашает к сотрудничеству авторов:
тел./факс: (812) 703-73-72, (495) 234-38-15; e-mail: ivanovaa@piter.com
Подробная информация здесь: http://www.piter.com/page/avtoru

Издательский дом «Питер» приглашает к сотрудничеству зарубежных торговых партнеров или посредников, имеющих выход на зарубежный рынок: *тел./факс: (812) 703-73-73; e-mail: sales@piter.com*

Заказ книг для вузов и библиотек:
тел./факс: (812) 703-73-73, доб. 6243; e-mail: uchebnik@piter.com

Заказ книг по почте: *на сайте www.piter.com; тел.: (812) 703-73-74, доб. 6216; e-mail: books@piter.com*

Вопросы по продаже электронных книг: *тел.: (812) 703-73-74, доб. 6217; e-mail: kuznetsov@piter.com*

КНИГА-ПОЧТОЙ

ЗАКАЗАТЬ КНИГИ ИЗДАТЕЛЬСКОГО ДОМА «ПИТЕР» МОЖНО ЛЮБЫМ УДОБНЫМ ДЛЯ ВАС СПОСОБОМ:

- на нашем сайте: **www.piter.com**
- по электронной почте: **books@piter.com**
- по телефону: **(812) 703-73-74**

ВЫ МОЖЕТЕ ВЫБРАТЬ ЛЮБОЙ УДОБНЫЙ ДЛЯ ВАС СПОСОБ ОПЛАТЫ:

Наложенным платежом с оплатой при получении в ближайшем почтовом отделении.

С помощью банковской карты. Во время заказа вы будете перенаправлены на защищенный сервер нашего оператора, где сможете ввести свои данные для оплаты.

Электронными деньгами. Мы принимаем к оплате Яндекс.Деньги, Webmoney и Kiwi-кошелек.

В любом банке, распечатав квитанцию, которая формируется автоматически после совершения вами заказа.

ВЫ МОЖЕТЕ ВЫБРАТЬ ЛЮБОЙ УДОБНЫЙ ДЛЯ ВАС СПОСОБ ДОСТАВКИ:

- Посылки отправляются через «Почту России». Отработанная система позволяет нам организовывать доставку ваших покупок максимально быстро. Дату отправления вашей покупки и дату доставки вам сообщат по e-mail.
- Вы можете оформить курьерскую доставку своего заказа (более подробную информацию можно получить на нашем сайте www.piter.com).
- Можно оформить доставку заказа через почтоматы (адреса почтоматов можно узнать на нашем сайте www.piter.com).

ПРИ ОФОРМЛЕНИИ ЗАКАЗА УКАЖИТЕ:

- фамилию, имя, отчество, телефон, e-mail;
- почтовый индекс, регион, район, населенный пункт, улицу, дом, корпус, квартиру;
- название книги, автора, количество заказываемых экземпляров.

БЕСПЛАТНАЯ ДОСТАВКА:

- курьером по Москве и Санкт-Петербургу при заказе на сумму **от 2000 руб.**
- почтой России при предварительной оплате заказа на сумму **от 2000 руб.**

ВАША УНИКАЛЬНАЯ КНИГА

Хотите издать свою книгу? Она станет идеальным подарком для партнеров и друзей, отличным инструментом для продвижения вашего бренда, презентом для памятных событий! Мы сможем осуществить ваши любые, даже самые смелые и сложные, идеи и проекты.

МЫ ПРЕДЛАГАЕМ:

- издать вашу книгу
- издание книги для использования в маркетинговых активностях
- книги как корпоративные подарки
- рекламу в книгах
- издание корпоративной библиотеки

Почему надо выбрать именно нас:

Издательству «Питер» более 20 лет. Наш опыт – гарантия высокого качества.

Мы предлагаем:

- услуги по обработке и доработке вашего текста
- современный дизайн от профессионалов
- высокий уровень полиграфического исполнения
- продажу вашей книги во всех книжных магазинах страны

Обеспечим продвижение вашей книги:

- рекламой в профильных СМИ и местах продаж
- рецензиями в ведущих книжных изданиях
- интернет-поддержкой рекламной кампании

Мы имеем собственную сеть дистрибуции по всей России, а также на Украине и в Беларуси. Сотрудничаем с крупнейшими книжными магазинами. Издательство «Питер» является постоянным участником многих конференций и семинаров, которые предоставляют широкую возможность реализации книг.

Мы обязательно проследим, чтобы ваша книга постоянно имелась в наличии в магазинах и была выложена на самых видных местах.

Обеспечим индивидуальный подход к каждому клиенту, эксклюзивный дизайн, любой тираж.

Кроме того, предлагаем вам выпустить электронную книгу. Мы разместим ее в крупнейших интернет-магазинах. Книга будет сверстана в формате ePub или PDF – самых популярных и надежных форматах на сегодняшний день.

Свяжитесь с нами прямо сейчас:

Санкт-Петербург – *Анна Титова, (812) 703-73-73, titova@piter.com*
Москва – *Сергей Клебанов, (495) 234-38-15, klebanov@piter.com*