O'REILLY®

Второе издание

Простой Python

Современный стиль программирования

Билл Любанович

SECOND EDITION

Introducing Python

Modern Computing in Simple Packages

Bill Lubanovic

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

Простой Python

Современный стиль программирования

Второе издание

Билл Любанович

ПИТЕР®

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

2021

ББК 32.973.2-018.1
УДК 004.43
Л93

Любанович Билл

Л93 Простой Python. Современный стиль программирования. 2-е изд. — СПб.: Питер, 2021. — 592 с.: ил. — (Серия «Бестселлеры O'Reilly»).

ISBN 978-5-4461-1639-3

«Простой Python» познакомит вас с одним из самых популярных языков программирования. Книга идеально подойдет как начинающим, так и опытным программистам, желающим добавить Python к списку освоенных языков.

Любому программисту нужно знать не только язык, но и его возможности. Вы начнете с основ Python и его стандартной библиотеки. Узнаете, как находить, загружать, устанавливать и использовать сторонние пакеты. Изучите лучшие практики тестирования, отладки, повторного использования кода и получите полезные советы по разработке. Примеры кода и упражнения помогут в создании приложений для различных целей.

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

ББК 32.973.2-018.1
УДК 004.43

ISBN 978-1492051367 англ.

ISBN 978-5-4461-1639-3

Краткое содержание

Введение 26
Благодарности 33
Об авторе 34

Часть I. Основы Python

Глава 1. Python: с чем его едят 36
Глава 2. Данные: типы, значения, переменные и имена 55
Глава 3. Числа 67
Глава 4. Выбираем с помощью оператора if 82
Глава 5. Текстовые строки 91
Глава 6. Создаем циклы с помощью ключевых слов while и for 113
Глава 7. Кортежи и списки 119
Глава 8. Словари и множества 144
Глава 9. Функции 166
Глава 10. Ой-ой-ой: объекты и классы 194
Глава 11. Модули, пакеты и программы 224

Часть II. Python на практике

Глава 12. Обрабатываем данные 242
Глава 13. Календари и часы 271
Глава 14. Файлы и каталоги 283
Глава 15. Данные во времени: процессы и конкурентность 302

Глава 16. Данные в коробке: надежные хранилища 327
Глава 17. Данные в пространстве: сети 368
Глава 18. Распутываем Всемирную паутину 400
Глава 19. Быть питонщиком 431
Глава 20. Пи-Арт 474
Глава 21. За работой 487
Глава 22. Python в науке 503

Приложения

Приложение А. Аппаратное и программное обеспечение для начинающих программистов 520
Приложение Б. Установка Python 3 530
Приложение В. Нечто совершенно иное: async 538
Приложение Г. Ответы к упражнениям 544
Приложение Д. Вспомогательные таблицы 587
Эпилог 591

Оглавление

Введение 26
Для кого эта книга 27
Что нового во втором издании 27
Структура книги 28
Версии Python 31
Условные обозначения 31
Использование примеров кода 32
От издательства 32

Благодарности 33

Об авторе 34

Часть I. Основы Python

Глава 1. Python: с чем его едят 36
Тайны 36
Маленькие программы 38
Более объемная программа 40
Python в реальном мире 44
Python против языка с планеты X 45
Почему же Python? 48
Когда не стоит использовать Python 49
Python 2 против Python 3 50
Установка Python 51
Запуск Python 51
Интерактивный интерпретатор 51
Файлы Python 52
Что дальше? 53

Момент просветления ... 53
Читайте далее ... 54
Упражнения ... 54

Глава 2. Данные: типы, значения, переменные и имена ... 55
В Python данные являются объектами ... 55
Типы ... 56
Изменчивость ... 57
Значения-литералы ... 58
Переменные ... 58
Присваивание ... 60
Переменные — это имена, а не локации ... 61
Присваивание нескольким именам ... 64
Переназначение имени ... 64
Копирование ... 64
Выбираем хорошее имя переменной ... 65
Читайте далее ... 66
Упражнения ... 66

Глава 3. Числа ... 67
Булевы значения ... 67
Целые числа ... 68
Числа-литералы ... 68
Операции с целыми числами ... 69
Целые числа и переменные ... 71
Приоритет операций ... 73
Системы счисления ... 74
Преобразования типов ... 76
Насколько объемен тип int ... 78
Числа с плавающей точкой ... 79
Математические функции ... 80
Читайте далее ... 81
Упражнения ... 81

Глава 4. Выбираем с помощью оператора if ... 82
Комментируем с помощью символа # ... 82
Продлеваем строки с помощью символа \ ... 83
Сравниваем с помощью операторов if, elif и else ... 84
Что есть истина? ... 87
Выполняем несколько сравнений с помощью оператора in ... 88

Новое: I Am the Walrus 89
Читайте далее 90
Упражнения 90

Глава 5. Текстовые строки 91
Создаем строки с помощью кавычек 91
Создаем строки с помощью функции str() 94
Создаем escape-последовательности с помощью символа \ 94
Объединяем строки с использованием символа + ... 96
Размножаем строки с помощью символа * 96
Извлекаем символ с помощью символов [ ] 97
Извлекаем подстроки, используя разделение 98
Измеряем длину строки с помощью функции len() 100
Разделяем строку с помощью функции split() 100
Объединяем строки с помощью функции join() 101
Заменяем символы с использованием функции replace() 101
Устраняем символы с помощью функции strip() 102
Поиск и выбор 103
Регистр 104
Выравнивание 105
Форматирование 105
Старый стиль: % 106
Новый стиль: используем символы {} и функцию format() 108
Самый новый стиль: f-строки 110
Что еще можно делать со строками 111
Читайте далее 111
Упражнения 111

Глава 6. Создаем циклы с помощью ключевых слов while и for 113
Повторяем действия с помощью цикла while 113
Прерываем цикл с помощью оператора break 114
Пропускаем итерации, используя оператор continue 114
Проверяем, завершился ли цикл раньше, с помощью блока else 115
Выполняем итерации с использованием ключевых слов for и in 115
Прерываем цикл с помощью оператора break 116
Пропускаем итерации, используя оператор continue 116
Проверяем, завершился ли цикл раньше, с помощью блока else 116
Генерируем числовые последовательности с помощью функции range() 117
Прочие итераторы 118
Читайте далее 118
Упражнения 118

Глава 7. Кортежи и списки ... 119
Кортежи ... 119
Создаем кортежи с помощью запятых и оператора () ... 120
Создаем кортежи с помощью функции tuple() ... 121
Объединяем кортежи с помощью оператора + ... 121
Размножаем элементы с помощью оператора * ... 122
Сравниваем кортежи ... 122
Итерируем по кортежам с помощью for и in ... 122
Изменяем кортеж ... 122
Списки ... 123
Создаем списки с помощью скобок [] ... 123
Создаем список или преобразуем в список с помощью функции list() ... 123
Создаем список из строки с использованием функции split() ... 124
Получаем элемент с помощью конструкции [смещение] ... 124
Извлекаем элементы с помощью разделения ... 125
Добавляем элемент в конец списка с помощью функции append() ... 126
Добавляем элемент на определенное место с помощью функции insert() ... 126
Размножаем элементы с помощью оператора * ... 127
Объединяем списки с помощью метода extend() или оператора + ... 127
Изменяем элемент с помощью конструкции [смещение] ... 128
Изменяем элементы с помощью разделения ... 128
Удаляем заданный элемент с помощью оператора del ... 129
Удаляем элемент по значению с помощью функции remove() ... 129
Получаем и удаляем заданный элемент с помощью функции pop() ... 129
Удаляем все элементы с помощью функции clear() ... 130
Определяем смещение по значению с помощью функции index() ... 130
Проверяем на наличие элемента в списке с помощью оператора in ... 131
Подсчитываем количество включений значения с помощью функции count() ... 131
Преобразуем список в строку с помощью функции join() ... 131
Меняем порядок элементов с помощью функций sort() или sorted() ... 132
Получаем длину списка с помощью функции len() ... 133
Присваиваем с помощью оператора = ... 133
Копируем списки с помощью функций copy() и list() или путем разделения ... 134
Копируем все с помощью функции deepcopy() ... 134
Сравниваем списки ... 135
Итерируем по спискам с помощью операторов for и in ... 136

Итерируем по нескольким последовательностям с помощью функции zip() ... 137
Создаем список с помощью списковых включений ... 138
Списки списков ... 140
Кортежи или списки? ... 141
Включений кортежей не существует ... 141
Читайте далее ... 142
Упражнения ... 142

Глава 8. Словари и множества ... 144
Словари ... 144
Создаем словарь с помощью {} ... 144
Создаем словарь с помощью функции dict() ... 145
Преобразуем с помощью функции dict() ... 146
Добавляем или изменяем элемент с помощью конструкции [ключ] ... 146
Получаем элемент словаря с помощью конструкции [ключ] или функции get() ... 148
Получаем все ключи с помощью функции keys() ... 148
Получаем все значения с помощью функции values() ... 149
Получаем все пары «ключ — значение» с помощью функции items() ... 149
Получаем длину словаря с помощью функции len() ... 149
Объединяем словари с помощью конструкции {**a, **b} ... 149
Объединяем словари с помощью функции update() ... 150
Удаляем элементы по их ключу с помощью оператора del ... 151
Получаем элемент по ключу и удаляем его с помощью функции pop() ... 151
Удаляем все элементы с помощью функции clear() ... 151
Проверяем на наличие ключа с помощью оператора in ... 152
Присваиваем значения с помощью оператора = ... 152
Копируем значения с помощью функции copy() ... 152
Копируем все с помощью функции deepcopy() ... 153
Сравниваем словари ... 154
Итерируем по словарям с помощью for и in ... 154
Включения словарей ... 155
Множества ... 156
Создаем множество с помощью функции set() ... 157
Преобразуем другие типы данных с помощью функции set() ... 157
Получаем длину множества с помощью функции len() ... 158
Добавляем элемент с помощью функции add() ... 158
Удаляем элемент с помощью функции remove() ... 158

Итерируем по множествам с помощью for и in ... 158
Проверяем на наличие значения с помощью оператора in ... 158
Комбинации и операторы ... 159
Включение множества ... 162
Создаем неизменяемое множество с помощью функции frozenset() ... 162
Структуры данных, которые мы уже рассмотрели ... 163
Создание крупных структур данных ... 164
Читайте далее ... 164
Упражнения ... 165

Глава 9. Функции ... 166
Определяем функцию с помощью ключевого слова def ... 166
Вызываем функцию с помощью скобок ... 167
Аргументы и параметры ... 167
None — это полезно ... 169
Позиционные аргументы ... 170
Аргументы — ключевые слова ... 171
Указываем значение параметра по умолчанию ... 171
Получаем/разбиваем аргументы — ключевые слова с помощью символа * ... 172
Получаем/разбиваем аргументы — ключевые слова с помощью символов ** ... 174
Аргументы, передаваемые только по ключевым словам ... 175
Изменяемые и неизменяемые аргументы ... 176
Строки документации ... 176
Функции — это объекты первого класса ... 177
Внутренние функции ... 179
Анонимные функции: лямбда-выражения ... 181
Генераторы ... 182
Функции-генераторы ... 182
Включения генераторов ... 183
Декораторы ... 183
Пространства имен и область определения ... 186
Использование символов _ и __ в именах ... 188
Рекурсия ... 188
Асинхронные функции ... 190
Исключения ... 190
Обрабатываем ошибки с помощью операторов try и except ... 191
Создаем собственные исключения ... 192
Читайте далее ... 193
Упражнения ... 193

Глава 10. Ой-ой-ой: объекты и классы ... 194
Что такое объекты ... 194
Простые объекты ... 195
Определяем класс с помощью ключевого слова class ... 195
Атрибуты ... 196
Методы ... 197
Инициализация ... 197
Наследование ... 198
Наследование от родительского класса ... 199
Переопределение методов ... 200
Добавление метода ... 201
Получаем помощь от своего родителя с использованием метода super() ... 202
Множественное наследование ... 203
Примеси ... 205
В защиту self ... 205
Доступ к атрибутам ... 206
Прямой доступ ... 206
Геттеры и сеттеры ... 206
Свойства для доступа к атрибутам ... 207
Свойства для вычисляемых значений ... 209
Искажение имен для безопасности ... 209
Атрибуты классов и объектов ... 210
Типы методов ... 211
Методы объектов ... 211
Методы классов ... 212
Статические методы ... 212
Утиная типизация ... 213
Магические методы ... 215
Агрегирование и композиция ... 218
Когда использовать объекты, а когда — что-то другое ... 218
Именованные кортежи ... 219
Классы данных ... 221
attrs ... 222
Читайте далее ... 222
Упражнения ... 222

Глава 11. Модули, пакеты и программы ... 224
Модули и оператор import ... 224
Импортируем модуль ... 224

Импортируем модуль с другим именем ... 226
Импортируем только самое необходимое ... 226
Пакеты ... 227
Путь поиска модуля ... 228
Относительный и абсолютный импорт ... 229
Пакеты пространств имен ... 229
Модули против объектов ... 230
Достоинства стандартной библиотеки Python ... 231
Обрабатываем отсутствующие ключи с помощью функций setdefault() и defaultdict() ... 231
Подсчитываем элементы с помощью функции Counter() ... 233
Упорядочиваем по ключу с помощью OrderedDict() ... 235
Стек + очередь == deque ... 235
Итерируем по структурам кода с помощью модуля itertools ... 236
Красиво выводим данные на экран с помощью функции pprint() ... 238
Работаем со случайными числами ... 238
Нужно больше кода ... 239
Читайте далее ... 240
Упражнения ... 240

Часть II. Python на практике

Глава 12. Обрабатываем данные ... 242
Текстовые строки: Unicode ... 243
Строки формата Unicode в Python 3 ... 244
Кодирование и декодирование с помощью кодировки UTF-8 ... 246
Кодирование ... 247
Декодирование ... 249
Сущности HTML ... 250
Нормализация ... 251
Подробная информация ... 252
Текстовые строки: регулярные выражения ... 253
Ищем точное начальное совпадение с помощью функции match() ... 254
Ищем первое совпадение с помощью функции search() ... 255
Ищем все совпадения, используя функцию findall() ... 255
Разбиваем совпадения с помощью функции split() ... 256
Заменяем совпадения с помощью функции sub() ... 256
Шаблоны: специальные символы ... 256
Шаблоны: использование спецификаторов ... 258
Шаблоны: указываем способ вывода совпадения ... 261

Бинарные данные ... 261
bytes и bytearray ... 262
Преобразуем бинарные данные с помощью модуля struct ... 263
Другие инструменты для работы с бинарными данными ... 266
Преобразуем байты/строки с помощью модуля binascii ... 267
Битовые операторы ... 267
Аналогия с ювелирными изделиями ... 268
Читайте далее ... 268
Упражнения ... 268

Глава 13. Календари и часы ... 271
Високосный год ... 272
Модуль datetime ... 273
Модуль time ... 275
Читаем и записываем дату и время ... 277
Все преобразования ... 281
Альтернативные модули ... 281
Читайте далее ... 282
Упражнения ... 282

Глава 14. Файлы и каталоги ... 283
Ввод информации в файлы и ее вывод из них ... 283
Создаем или открываем файлы с помощью функции open() ... 284
Записываем в текстовый файл с помощью функции print() ... 284
Записываем в текстовый файл с помощью функции write() ... 285
Считываем данные из текстового файла, используя функции read(), readline() и readlines() ... 286
Записываем данные в бинарный файл с помощью функции write() ... 288
Читаем бинарные файлы с помощью функции read() ... 289
Закрываем файлы автоматически с помощью ключевого слова with ... 289
Меняем позицию с помощью функции seek() ... 289
Отображение в памяти ... 291
Операции с файлами ... 292
Проверяем существование файла с помощью функции exists() ... 292
Проверяем тип с помощью функции isfile() ... 292
Копируем файлы, используя функцию copy() ... 293
Изменяем имена файлов с помощью функции rename() ... 293
Создаем ссылки с помощью функции link() или symlink() ... 293
Изменяем разрешения с помощью функции chmod() ... 294
Изменение владельца файла с помощью функции chown() ... 294
Удаляем файл с помощью функции remove() ... 294

Каталоги ... 295
Создаем каталог с помощью функции mkdir() ... 295
Удаляем каталог, используя функцию rmdir() ... 295
Выводим на экран содержимое каталога с помощью функции listdir() ... 295
Изменяем текущий каталог с помощью функции chdir() ... 296
Перечисляем совпадающие файлы, используя функцию glob() ... 296
Pathname ... 297
Получаем путь с помощью функции abspath() ... 298
Получаем символьную ссылку с помощью функции realpath() ... 298
Построение пути с помощью функции os.path.join() ... 298
Модуль pathlib ... 298
BytesIO и StringIO ... 299
Читайте далее ... 301
Упражнения ... 301

Глава 15. Данные во времени: процессы и конкурентность ... 302
Программы и процессы ... 302
Создаем процесс с помощью модуля subprocess ... 303
Создаем процесс с помощью модуля multiprocessing ... 304
Убиваем процесс, используя функцию terminate() ... 305
Получаем системную информацию с помощью модуля os ... 306
Получаем информацию о процессах с помощью модуля psutil ... 306
Автоматизация команд ... 307
Invoke ... 307
Другие вспомогательные методы для команд ... 308
Конкурентность ... 308
Очереди ... 309
Процессы ... 310
Потоки ... 311
Concurrent.futures ... 314
Зеленые потоки и gevent ... 317
twisted ... 320
asyncio ... 321
Redis ... 321
Помимо очередей ... 325
Читайте далее ... 326
Упражнения ... 326

Глава 16. Данные в коробке: надежные хранилища ... 327
Плоские текстовые файлы ... 327
Текстовые файлы, дополненные пробелами ... 328

Структурированные текстовые файлы ... 328
CSV ... 328
XML ... 331
Примечание о безопасности XML ... 333
HTML ... 333
JSON ... 334
YAML ... 337
Tablib ... 338
Pandas ... 338
Конфигурационные файлы ... 340
Бинарные файлы ... 341
Электронные таблицы ... 341
HDF5 ... 341
TileDB ... 342
Реляционные базы данных ... 342
SQL ... 343
DB-API ... 345
SQLite ... 345
MySQL ... 347
PostgreSQL ... 347
SQLAlchemy ... 348
Другие пакеты для работы с базами данных ... 354
Хранилища данных NoSQL ... 354
Семейство dbm ... 354
Memcached ... 355
Redis ... 356
Документоориентированные базы данных ... 363
Базы данных временных рядов ... 364
Графовые базы данных ... 365
Другие серверы NoSQL ... 365
Полнотекстовые базы данных ... 366
Читайте далее ... 366
Упражнения ... 366

Глава 17. Данные в пространстве: сети ... 368
TCP/IP ... 368
Сокеты ... 370
scapy ... 374
Netcat ... 374

Паттерны для работы с сетями ... 375
Паттерн «Запрос — ответ» ... 375
ZeroMQ ... 375
Другие инструменты обмена сообщениями ... 380
Паттерн «Публикация — подписка» ... 380
Redis ... 380
ZeroMQ ... 382
Другие инструменты «Публикации — подписки» ... 383
Интернет-сервисы ... 384
Доменная система имен ... 384
Модули Python для работы с электронной почтой ... 385
Другие протоколы ... 385
Веб-сервисы и API ... 385
Сериализация данных ... 386
Сериализация с помощью pickle ... 387
Другие форматы сериализации ... 388
Удаленные вызовы процедур ... 388
XML RPC ... 389
JSON RPC ... 390
MessagePack RPC ... 391
Zerorpc ... 392
gRPC ... 393
Twirp ... 393
Инструменты удаленного управления ... 394
Работаем с большими объемами данных ... 394
Hadoop ... 394
Spark ... 395
Disco ... 395
Dask ... 395
Работаем в облаках ... 396
Amazon Web Services ... 397
Google ... 397
Microsoft Azure ... 397
OpenStack ... 398
Docker ... 398
Kubernetes ... 398
Читайте далее ... 398
Упражнения ... 399

Глава 18. Распутываем Всемирную паутину ... 400
Веб-клиенты ... 401
Тестируем с помощью telnet ... 402
Тестируем с помощью curl ... 403
Тестируем с использованием httpie ... 404
Тестируем с помощью httpbin ... 405
Стандартные веб-библиотеки Python ... 405
За пределами стандартной библиотеки: requests ... 407
Веб-серверы ... 408
Простейший веб-сервер Python ... 409
Web Server Gateway Interface (WSGI) ... 410
ASGI ... 411
apache ... 411
NGINX ... 412
Другие веб-серверы Python ... 413
Фреймворки для работы веб-серверами ... 413
Bottle ... 414
Flask ... 416
Django ... 420
Другие фреймворки ... 421
Фреймворки для работы с базами данных ... 421
Веб-сервисы и автоматизация ... 422
Модуль webbrowser ... 422
Модуль webview ... 423
REST API ... 424
Поиск и выборка данных ... 424
Scrapy ... 425
BeautifulSoup ... 425
Requests-HTML ... 426
Давайте посмотрим фильм ... 426
Читайте далее ... 429
Упражнения ... 429

Глава 19. Быть питонщиком ... 431
О программировании ... 431
Ищем код на Python ... 432
Установка пакетов ... 432
pip ... 433
virtualenv ... 434

pipenv ... 434
Менеджер пакетов ... 434
Установка из исходного кода ... 435
Интегрированные среды разработки ... 435
IDLE ... 435
PyCharm ... 435
IPython ... 436
Jupyter Notebook ... 438
JupyterLab ... 438
Именование и документирование ... 438
Добавление подсказок типов ... 440
Тестирование кода ... 440
Программы pylint, pyflakes, flake8 или PEP-8 ... 441
Пакет unittest ... 443
Пакет doctest ... 447
Пакет nose ... 448
Другие фреймворки для тестирования ... 449
Постоянная интеграция ... 449
Отладка кода ... 450
Функция print() ... 450
Отладка с помощью декораторов ... 451
Отладчик pdb ... 452
Функция breakpoint() ... 458
Записываем в журнал сообщения об ошибках ... 458
Оптимизация кода ... 460
Измеряем время ... 461
Алгоритмы и структуры данных ... 464
Cython, NumPy и расширения C ... 465
PyPy ... 465
Numba ... 466
Управление исходным кодом ... 467
Mercurial ... 467
Git ... 467
Распространение ваших программ ... 470
Клонируйте эту книгу ... 470
Как узнать больше ... 470
Книги ... 471
Сайты ... 471
Группы ... 472

Конференции ... 472
Вакансии, связанные с Python ... 472
Читайте далее ... 473
Упражнения ... 473

Глава 20. Пи-Арт ... 474
Двумерная графика ... 474
Стандартная библиотека ... 474
PIL и Pillow ... 475
ImageMagick ... 478
Трехмерная графика ... 478
Трехмерная анимация ... 479
Графические пользовательские интерфейсы (GUI) ... 479
Диаграммы, графики и визуализация ... 481
Matplotlib ... 481
Seaborn ... 483
Bokeh ... 485
Игры ... 485
Аудио и музыка ... 486
Читайте далее ... 486
Упражнения ... 486

Глава 21. За работой ... 487
The Microsoft Office Suite ... 487
Выполняем бизнес-задачи ... 488
Обработка бизнес-данных ... 489
Извлечение, преобразование и загрузка ... 489
Валидация данных ... 493
Дополнительные источники информации ... 493
Пакеты для работы с бизнес-данными с открытым исходным кодом ... 494
Python в области финансов ... 494
Безопасность бизнес-данных ... 495
Карты ... 495
Форматы ... 496
Нарисуем карту на основе шейп-файла ... 496
Geopandas ... 498
Другие пакеты для работы с картами ... 500
Приложения и данные ... 501
Читайте далее ... 502
Упражнения ... 502

Глава 22. Python в науке ... 503
Математика и статистика в стандартной библиотеке ... 503
Математические функции ... 503
Работа с комплексными числами ... 505
Рассчитываем точное значение чисел с плавающей точкой с помощью модуля decimal ... 506
Выполняем вычисления для рациональных чисел с помощью модуля fractions ... 507
Используем Packed Sequences с помощью модуля array ... 507
Обрабатываем простую статистику с помощью модуля statistics ... 508
Перемножение матриц ... 508
Python для науки ... 508
NumPy ... 508
Создаем массив с помощью функции array() ... 509
Создаем массив с помощью функции arange() ... 510
Создаем массив с помощью функций zeros(), ones() и random() ... 511
Изменяем форму массива с помощью метода reshape() ... 512
Получаем элемент с помощью конструкции [] ... 513
Математика массивов ... 514
Линейная алгебра ... 514
Библиотека SciPy ... 515
Библиотека SciKit ... 516
Pandas ... 516
Python и научные области ... 517
Читайте далее ... 518
Упражнения ... 518

Приложения

Приложение А. Аппаратное и программное обеспечение для начинающих программистов ... 520
Аппаратное обеспечение ... 520
Компьютеры пещерных людей ... 520
Электричество ... 521
Изобретения ... 521
Идеальный компьютер ... 522
Процессор ... 522
Память и кэш ... 522
Хранение ... 522
Ввод данных ... 523

Вывод данных ... 523
Относительное время доступа ... 523
Программное обеспечение ... 524
Вначале был бит ... 524
Машинный язык ... 524
Ассемблер ... 525
Высокоуровневые языки ... 525
Операционные системы ... 526
Виртуальные машины ... 527
Контейнеры ... 527
Распределенные вычисления и сети ... 527
Облако ... 528
Kubernetes ... 528

Приложение Б. Установка Python 3 ... 530
Проверьте свою версию Python ... 530
Установка стандартной версии Python ... 531
macOS ... 532
Windows ... 534
Linux или Unix ... 535
Установка менеджера пакетов pip ... 535
Установка virtualenv ... 535
Другие способы работы с пакетами ... 536
Устанавливаем Anaconda ... 536

Приложение В. Нечто совершенно иное: async ... 538
Сопрограммы и циклы событий ... 538
async против... ... 542
Асинхронные фреймворки и серверы ... 542

Приложение Г. Ответы к упражнениям ... 544
1. Python: с чем его едят ... 544
2. Типы данных, значения, переменные и имена ... 545
3. Числа ... 545
4. Выбираем с помощью if ... 546
5. Текстовые строки ... 547
6. Создаем циклы с помощью ключевых слов while и for ... 551
7. Кортежи и списки ... 552
8. Словари и множества ... 556
9. Функции ... 559
10. Ой-ой-ой: объекты и классы ... 560

11. Модули, пакеты и программы ... 564
12. Обрабатываем данные ... 566
13. Календари и часы ... 571
14. Файлы и каталоги ... 572
15. Данные во времени: процессы и конкурентность ... 573
16. Данные в коробке: устойчивые хранилища ... 574
17. Данные в пространстве: сети ... 577
18. Распутываем Всемирную паутину ... 584
19. Быть питонщиком ... 585
20. Пи-Арт ... 585
21. За работой ... 586
22. Python в науке ... 586

Приложение Д. Вспомогательные таблицы ... 587
Приоритет операторов ... 587
Строковые методы ... 588
Изменение регистра ... 588
Поиск ... 588
Изменение ... 588
Форматирование ... 589
Тип строки ... 589
Атрибуты модуля string ... 589

Эпилог ... 591

С любовью к Мэри, Тому и Рокси,
а также Кэрин и Эрику.

Введение

Как и обещает название, книга познакомит вас с одним из самых популярных языков программирования — Python. Издание предназначено как для начинающих программистов, так и для тех, кто уже имеет опыт в написании программ и просто желает добавить Python к списку доступных ему языков.

В большинстве случаев изучать компьютерный язык проще, чем человеческий, — в нем меньше двусмысленностей и исключений, которые приходится запоминать. Python — один из самых последовательных и понятных компьютерных языков, он сочетает в себе простоту изучения, простоту использования и большую выразительную силу.

Компьютерные языки состоят из *данных* — аналогами в разговорной речи являются существительные — и *инструкций* (или *кода*), которые можно сравнить с глаголами. Изучить нужно будет и то и другое: вы освоите основы кода и структур данных и узнаете, как их объединить. Затем перейдете к более сложным темам, а программы, которые вы будете читать и писать, станут длиннее и сложнее. Если провести аналогию с работой по дереву, мы начнем с молотка, гвоздей и небольших кусков древесины, а во второй половине книги обратимся к более специализированным инструментам, которые можно сравнить с токарными станками и другими более сложными устройствами.

Вам нужно знать не только сам язык, но и то, что с его помощью можно делать. Мы начнем с языка Python и его стандартной библиотеки, готовой к работе прямо «из коробки». Помимо этого, я покажу вам, как находить, загружать, устанавливать и использовать качественные сторонние пакеты: касаться узких тем или рассматривать сложные трюки не буду, а сделаю акцент на том, что после десяти лет работы с Python считаю действительно полезным.

Несмотря на то что книга представляет собой введение в Python, в ней мы затронем и несколько дополнительных тем, с которыми, на мой взгляд, следует ознакомиться еще на начальном этапе. Работе с базами данных и Интернетом мы тоже уделим внимание, однако не станем забывать, что технологии меняются очень быстро — теперь от программиста на Python общество ждет знаний об облачных технологиях, машинном обучении и создании потоков событий: основную информацию по этим темам вы также здесь найдете.

Язык Python имеет некоторые специальные функции, работающие лучше, чем адаптированные стили программирования из других языков. Например, использование ключевого слова `for` и *итераторов* — более прямой способ создания

цикла: пользоваться им гораздо удобнее, нежели вручную инкрементировать переменную-счетчик.

Когда вы изучаете что-то новое, бывает трудно определиться с тем, какие слова являются терминами и какие понятия на самом деле важны. Иначе говоря, тестировалась ли эта функциональность? Я выделю некоторые термины и понятия, которые имеют особое значение в Python. Код, написанный на языке Python, можно будет увидеть даже в самых первых главах.

Подобные примечания я буду делать, когда что-то может быть непонятным или же существует более питонский способ решить проблему.

Python неидеален. Я обращу ваше внимание на то, что кажется сомнительным и чего следует избегать, и предложу альтернативные варианты.

По некоторым темам, таким как наследование объектов, MVC или проектирование REST для работы с Интернетом, мое мнение может отличаться от общепринятого. Вам самим решать, к кому прислушаться.

Для кого эта книга

Эта книга для всех, кто заинтересован в изучении одного из самых популярных во всем мире языков программирования. Наличие или отсутствие опыта с другими языками программирования не имеет значения.

Что нового во втором издании

Что изменилось со времени выхода первого издания?

- Добавилось около 100 страниц, в том числе с изображением котиков.
- Количество глав удвоилось, но сами главы стали короче.
- В начале книги появилась глава, посвященная типам данных, переменным и именам.
- Рассмотрены новые особенности Python, такие как *f-строки*.
- Рекомендованы новые или улучшенные сторонние библиотеки.
- Во всей книге присутствуют новые примеры кода.
- Для начинающих разработчиков добавлен текст с описанием аппаратного и программного обеспечения.
- Более опытные разработчики могут ознакомиться с библиотекой *asyncio*.
- Рассмотрен новый стек технологий: контейнеры, облачные технологии, наука о данных и машинное обучение.

- Добавлены подсказки, с помощью которых вы сможете найти работу программиста на Python.

Что не изменилось? Примеры, в которых используются плохие стихотворения и утки. Они с нами навсегда.

Структура книги

В первой части излагаются основы языка программирования Python: главы 1–11 следует читать по порядку. Я оперирую простейшими структурами данных и кода, постепенно составляя из них более сложные и реалистичные программы. Во второй части (главы 12–22) показывается, каким образом язык программирования Python используется в определенных прикладных областях, таких как Интернет, базы данных, сети и т. д.: эти главы можно читать в любом порядке.

Вот краткое содержание всех глав и приложений и обзор новых терминов, с которыми вы там встретитесь.

- *Глава 1. «Python: с чем его едят»*. Компьютерные программы не так уж и отличаются от других инструкций, с которыми вы сталкиваетесь каждый день. Мы рассмотрим небольшие программы, написанные на Python. Они продемонстрируют синтаксис языка, его возможности и способы применения в реальном мире. Вы узнаете, как запустить программу внутри *интерактивного интерпретатора (оболочки)*, а также из текстового *файла*, сохраненного на вашем компьютере.
- *Глава 2. «Данные: типы, значения, переменные и имена»*. В компьютерных языках используются данные и инструкции. Компьютер по-разному хранит и обрабатывает разные *типы* данных. Их значения или можно изменять (такие типы называются *изменяемыми*), или нельзя (*неизменяемые* типы). В программе, написанной на Python, данные могут быть представлены как литералами (числами вроде 78 или текстовыми *строками* вроде `"waffle"`), так и именованными *переменными*. В отличие от многих других языков программирования Python относится к переменным как к *именам*, и это влечет за собой некоторые важные последствия.
- *Глава 3. «Числа»*. В этой главе показываются простейшие типы данных, применяемые в языке программирования Python: *булевы переменные*, *целые числа* и *числа с плавающей точкой*. Вы также изучите простейшую математику. В примерах этой главы интерактивный интерпретатор Python используется как калькулятор.
- *Глава 4. «Выбираем с помощью оператора if»*. С существительными (типами данных) и с глаголами (программными структурами) мы поработаем в нескольких главах. Код, написанный на Python, обычно выполняется по одной строке за раз: от начала программы до ее конца. Структура `if` позволяет запускать разные строки кода исходя из результата сравнения определенных данных.
- *Глава 5. «Текстовые строки»*. Здесь мы обратимся к существительным и миру текстовых *строк*. Вы научитесь создавать, объединять, изменять и получать строки, а также выводить их на экран.

- *Глава 6. «Создаем циклы с помощью ключевых слов while и for».* Снова глаголы. Вы научитесь создавать *цикл* двумя способами — с помощью `for` и с помощью `while`, а также узнаете, что такое *итераторы* — одно из основных понятий Python.
- *Глава 7. «Кортежи и списки».* Пришло время рассмотреть первые структуры данных более высокого уровня: списки и кортежи. Они представляют собой последовательности значений, которыми вы будете пользоваться как конструктором Lego для того, чтобы создавать более сложные структуры. Вы научитесь *проходить* по ним с помощью *итераторов*, а также быстро создавать списки с помощью *списковых включений*.
- *Глава 8. «Словари и множества».* Словари и множества позволяют сохранять данные не по позиции, а по их значению. Вы увидите, насколько это удобно, — данная особенность Python станет одной из ваших любимых.
- *Глава 9. «Функции».* Соединяйте структуры данных из предыдущих глав со структурами кода, чтобы выполнять сравнение, выборку или повторение операций. Упаковывайте код в функции и обрабатывайте ошибки с помощью *исключений*.
- *Глава 10. «Ой-ой-ой: объекты и классы».* Слово «объект» недостаточно конкретно, но имеет большое значение во многих компьютерных языках, в том числе и в Python. Если вы уже занимались объектно-ориентированным программированием на других языках, то в сравнении с ними Python покажется вам более простым. В этой главе объясняется, когда следует использовать объекты и классы, а когда лучше выбрать другой путь.
- *Глава 11. «Модули, пакеты и программы».* Вы узнаете, как перейти к более крупным структурам кода — *модулям*, *пакетам* и *программам*, а также где можно разместить код и данные, как ввести и вывести данные, обработать различные параметры, просмотреть стандартную библиотеку Python и то, что находится вне ее.
- *Глава 12. «Обрабатываем данные».* Вы научитесь профессионально обрабатывать данные и управлять ими. Эта глава полностью посвящена текстовым и двоичным данным, особенностям использования символов стандарта Unicode, а также поиску текста с помощью *регулярных* выражений. Вы познакомитесь с типами данных `byte` и `bytearray` — соперниками типа `string`, в которых содержатся необработанные бинарные значения вместо текстовых символов.
- *Глава 13. «Календари и часы».* С датой и временем работать бывает непросто. Здесь мы рассмотрим распространенные проблемы и способы их решения.
- *Глава 14. «Файлы и каталоги».* Простые хранилища данных используют *файлы* и *каталоги*. В этой главе речь пойдет о создании и использовании файлов и каталогов.
- *Глава 15. «Данные во времени: процессы и конкурентность».* Это первая глава, в которой мы приступаем к изучению системы. Начнем с данных во времени — вы научитесь использовать *программы*, *процессы* и *потоки* для того, чтобы выполнять больше работы за один промежуток времени *(конкурентность)*. Среди прочего будут упомянуты последние добавления в *async* (более подробно они рассматриваются в приложении B).

- *Глава 16. «Данные в коробке: надежные хранилища»*. Данные могут храниться в простых файлах и каталогах внутри файловых систем и структурироваться с помощью распространенных форматов, таких как CSV, JSON и XML. Однако по мере того, как объем и сложность данных будут расти, вам, возможно, придется использовать *базы данных* — как традиционные *реляционные*, так и современные базы данных *NoSQL*.
- *Глава 17. «Данные в пространстве: сети»*. Отправляйте ваш код и данные через пространство по *сетям* с помощью *служб*, *протоколов* и *API*. В качестве примеров рассматриваются как низкоуровневые *сокеты*, библиотеки *обмена сообщениями* и системы массового обслуживания, так и развертывание в *облачных* системах.
- *Глава 18. «Распутываем Всемирную паутину»*. Всемирной сети посвящена отдельная глава, в которой рассматриваются клиенты, серверы, извлечение данных, API и фреймворки. Вы научитесь *искать* сайты и *извлекать* из них данные, а затем разработаете реальный сайт, используя параметры *запросов* и *шаблоны*.
- *Глава 19. «Быть питонщиком»*. В этой главе содержатся советы для программистов, пишущих на Python: вы получите рекомендации по установке (с помощью `pip` и `virtualenv`), использованию IDE, тестированию, отладке, журналированию, контролю исходного кода и документации. Узнаете также, как найти и установить полезные пакеты сторонних разработчиков, как упаковать свой код для повторного использования и где получить более подробную информацию.
- *Глава 20. «Пи-Арт»*. При помощи языка программирования Python можно создавать произведения искусства: в графике, музыке, анимации и играх.
- *Глава 21. «За работой»*. У Python есть специальные приложения для бизнеса: визуализация данных (графики, графы и карты), безопасность и регулирование.
- *Глава 22. «Python в науке»*. За последние несколько лет Python стал главным языком науки, он используется в математике, статистике, физике, биологии и медицине. Его сильные стороны — *наука о данных* и *машинное обучение*. В этой главе демонстрируются возможности таких инструментов, как NumPy, SciPy и Pandas.
- *Приложение А. «Аппаратное и программное обеспечение для начинающих программистов»*. Если вы новичок в мире программирования, из этого приложения вы можете узнать, как на самом деле работает аппаратное и программное обеспечение и что означают некоторые термины, с которыми в дальнейшем вам придется сталкиваться.
- *Приложение Б. «Установка Python 3»*. 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 другой вариант Unix.
- *Приложение В. «Нечто совершенно иное: async»*. В разных релизах Python добавляется функциональность для работы с асинхронностью — разобраться с ней может быть сложно. Я упоминаю о ней в тех главах, в которых заходит речь об асинхронности, но в этом приложении рассматриваю тему более подробно.
- *Приложение Г. «Ответы к упражнениям»*. Здесь содержатся ответы на упражнения, приведенные в конце каждой главы. Не подглядывайте туда, пока не по-

пробуете решить задачи самостоятельно, в противном случае вы рискуете превратиться в козленочка.

- *Приложение Д. «Вспомогательные таблицы»*. В этом приложении содержатся справочные данные.

Версии Python

Языки программирования со временем изменяются — разработчики добавляют в них новые возможности и исправляют ошибки. Примеры этой книги написаны и протестированы для версии Python 3.7. Версия 3.7 является наиболее современной на момент выхода этой книги, и о самых значимых нововведениях я расскажу. Версия 3.8 вышла в конце 2019 года — я рассмотрю самую ожидаемую функциональность[1]. Узнать, что и когда было добавлено в язык программирования Python, можно, посетив страницу https://docs.python.org/3/whatsnew/: там представлена техническая информация. Она, скорее всего, 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.

Условные обозначения

В этой книге приняты следующие шрифтовые соглашения.

Курсив

Им обозначаются новые термины и понятия.

`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.

Так оформлены совет, предложение или замечание.

Таким образом оформлено предупреждение.

[1] Оригинальное издание выпущено до выхода версии 3.8. Текущая версия — 3.8.2. — *Примеч. ред.*

Использование примеров кода

Примеры кода и упражнения, приведенные в тексте, доступны для загрузки по адресу https://github.com/madscheme/introducing-python. Эта книга написана, чтобы помочь вам в работе: вы можете применить код, содержащийся в ней, в ваших программах и документации и не связываться с нами, чтобы спросить 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, разрешение на это получить необходимо. Если вы цитируете это издание, отвечая на вопрос, или воспроизводите код из него в качестве примера, разрешения не требуется. Но если 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: «Простой Python. Современный стиль программирования. 2-е изд. Билл Любанович. Питер, 2020. 978-5-4461-1639-3».

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

От издательства

Ваши замечания, предложения, вопросы отправляйте по адресу comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

На веб-сайте издательства www.piter.com вы найдете подробную информацию о наших книгах.

Благодарности

Искренне благодарю обозревателей и читателей, помогавших сделать эту книгу лучше: Корбина Коллинза, Чарльза Гивра, Нейтана Стокса, Дейва Джорджа и Майка Джеймса.

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. Кроме того:

- 1982–1988 (Intran) — разрабатывал приложение Metaform на первой коммерческой графической рабочей станции;
- 1990–1995 (Northwest Airlines) — написал визуальную систему управления доходами, а также первый тест для анализа онлайн-маркетинга в Сети;
- 1994 (Tela) — стал сооснователем одного из первых интернет-провайдеров;
- 1995–1999 (WAM!NET) — разрабатывал веб-доски сообщений и 3M Digital Media Repository;
- 1999–2005 (Mad Scheme) — стал сооснователем компании, занимающейся веб-разработкой и хостингом;
- 2005 (O'Reilly) — в соавторстве написал несколько глав книги *Linux Server Security*[1];
- 2007 (O'Reilly) — в соавторстве написал книгу *Linux System Administration*[2];
- 2010–2013 (Keep) — разработал и построил службы Core Services, соединяющие веб-фронтенды и бэкенды, работающие с базами данных;
- 2014 (O'Reilly) — написал книгу *Introducing Python* (первое издание)[3];
- 2015–2016 (Internet Archive) — работал над API и адаптацией Wayback Machine на Python;
- 2016–2018 (CrowdStrike) — управлял сервисами на базе Python, которые обрабатывают миллиарды ежедневных событий, связанных с безопасностью.

Билл счастливо живет в горах Сангре-де-Сасквоч в штате Миннесота со своей чудесной семьей: женой Мэри, сыном Томом (и его женой Рокси) и дочерью Кэрин (и ее мужем Эриком), ухаживает за кошками Ингой и Люси и котом Честером.

---

1 *Bauer M.* Linux Server Security, 2nd Edition. — O'Reilly, 2009.

2 *Адельштайн Т., Любанович Б.* Системное администрирование в Linux. — СПб.: Питер, 2010.

3 *Любанович Б.* Простой Python. Современный стиль программирования. — СПб.: Питер, 2019.

ЧАСТЬ I

Основы Python

ГЛАВА 1
Python: с чем его едят

> Популярными становятся только уродливые языки. Python — исключение из этого правила.
>
> *Дональд Кнут*

Тайны

Начнем с двух небольших тайн и их разгадок. 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 некая компьютерная программа. На самом деле это *схема для вязания* — точнее, фрагмент, который описывает, как связать пятку носка. Похожие носки показаны на рис. 1.1.

Рис. 1.1. Вязаные носки

Для меня эти строки имеют не больше смысла, чем судоку для одного из моих котов, но вот моя жена совершенно точно понимает написанное. Если вы вяжете, то тоже поймете.

Рассмотрим еще один таинственный текст, который можно увидеть записанным на листочке из блокнота. Вы сразу поймете его предназначение, даже если и не догадаетесь о том, каким будет конечный продукт:

```
  1/2 столовой ложки масла или маргарина;
  1/2 столовой ложки сливок;
  2 1/2 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:
  присыпьте разделочную доску мукой;
  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 (рис. 1.2). Полейте блюдо маслом, вареньем или чем-либо еще, сверните и наслаждайтесь.

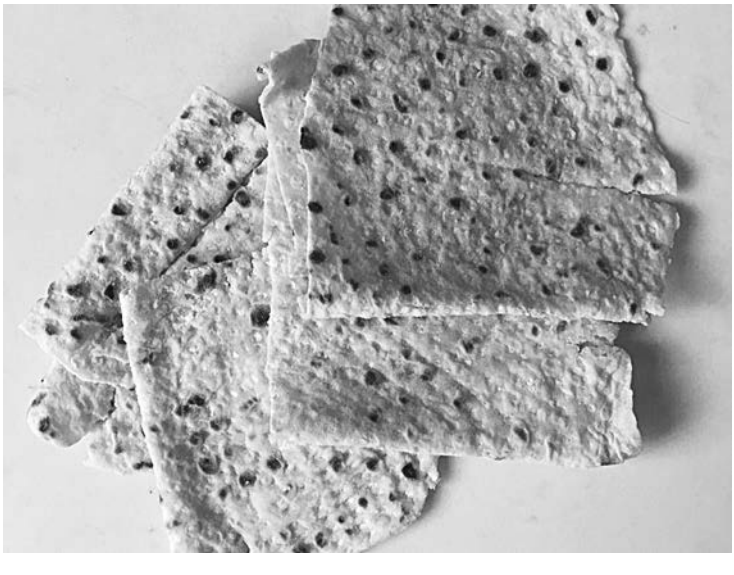

Рис. 1.2. Лефсе

Схема для вязания и рецепт имеют несколько схожих моментов:

- фиксированный *словарь*, состоящий из слов, аббревиатур и символов: какие-то могут быть вам знакомы, какие-то — нет;
- правила, описывающие, что и где можно говорить, — *синтаксис*;
- *последовательность операций*, которые должны быть выполнены в определенном порядке;
- в некоторых случаях — повторение определенных операций *(цикл)*, например способ приготовления каждого кусочка лефсе;
- в некоторых случаях — ссылка на еще одну последовательность операций (говоря компьютерными терминами, *функцию*). Например, когда вы прочтете приведенный выше рецепт, вам может понадобиться рецепт приготовления картофельного пюре;
- предполагаемое знание *контекста*. Рецепт подразумевает ваше знание о том, что такое вода и как ее кипятить. Схема для вязания подразумевает, что вы умеете держать спицы в руках;
- кое-какие *данные*, которые нужно использовать, создать или изменить, — картофель и нитки;
- *инструменты*, которые используются для работы с данными, — горшки, миксеры, духовки, вязальные спицы;
- ожидаемый *результат*. В наших примерах результатом будет предмет для ног и предмет для желудка. Главное — не перепутать.

Как ни назови — идиомы, жаргон, — примеры их использования можно встретить везде. Жаргон помогает сэкономить время тем, кто его знает, а для других людей оставляет информацию совершенно непонятной. Попробуйте расшифровать колонку газеты, посвященную бриджу, если вы не играете в эту игру, или научную статью — если вы не ученый (или ученый, но в другой области).

Маленькие программы

Подобные идеи вы встретите и в компьютерных программах, которые сами по себе являются маленькими языками: через них люди говорят компьютеру, что делать. Схему для вязания и рецепт я использовал для демонстрации того, что программы не так страшны, как может показаться, — всего лишь нужно выучить верные слова и правила.

Понять этот маленький язык гораздо легче, если в нем не очень много слов и правил и если вам не нужно изучать их все одновременно: за один раз наш мозг может воспринять только ограниченное количество знаний.

Пришло время обратиться к настоящей программе (пример 1.1). Как вы думаете, что она делает?

Пример 1.1. countdown.py

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

Программа, написанная на языке программирования Python, содержит некоторое количество специальных слов и символов: `for`, `in`, `print`, запятые, точки с запятой, скобки и т. д. — все они являются важной частью *синтаксиса* (правил) языка. Хорошая новость заключается в том, что Python имеет более доступный и менее объемный синтаксис по сравнению с большинством других языков программирования: текст кажется почти понятным — как и рецепт.

Пример 1.2 — тоже небольшая программа на Python: она позволяет выбрать одно из заклинаний Гарри Поттера, хранящееся в *списке*, и вывести его на экран.

Пример 1.2. spells.py

```
spells = [
    "Riddikulus!",
    "Wingardium Leviosa!",
    "Avada Kedavra!",
    "Expecto Patronum!",
    "Nox!",
    "Lumos!",
    ]
print(spells[3])
```

Отдельные заклинания являются в Python *строками* (последовательностями текстовых символов, заключенных в кавычки). Они разделены запятыми и помещены в *список* — это можно определить по квадратным скобкам (`[` и `]`). Слово `spells` — это *переменная*, являющаяся именем списка, — с ее помощью мы можем работать со списком. В нашем случае на экран будет выведено четвертое заклинание:

```
Expecto Patronum!
```

Почему мы сказали `3`, если нам нужно было четвертое заклинание? Списки Python, такие как `spells`, 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 обычно считают с единицы, поэтому считать с нуля может показаться странным. Однако в программировании удобнее оперировать смещениями, а не позициями. Да, это пример того, как компьютерная программа иногда отличается от обычного языка.

Список — очень распространенная *структура данных* в языке программирования Python. О том, как им пользоваться, будет рассказано в главе 7.

Программа из примера 1.3 выводит на экран цитату одного из участников комедийного трио The Three Stooges («Три балбеса»), однако на выбор фразы влияет не позиция в списке, а то, кто ее сказал.

Пример 1.3. quotes.py

```
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` — переменная, которая именует *словарь* Python: коллекцию уникальных *ключей* (в примере ключом является имя участника трио) и связанных с ними *значений* (в нашем примере — значимое высказывание участника «Балбесов»). Используя словарь, вы можете сохранять элементы и выполнять их поиск по именам: зачастую это удобнее, чем работать со списком.

В примере с заклинаниями для создания списка использовались квадратные скобки (`[` и `]`), а в примере с цитатами для создания словаря — фигурные скобки (`{` и `}`). Также мы использовали двоеточие (`:`) для того, чтобы связать каждый ключ словаря с соответствующим значением. Более подробно о словарях можно прочитать в главе 8.

Надеюсь, я не перегрузил вас синтаксисом. В следующих нескольких разделах вы познакомитесь и с другими простыми правилами.

Более объемная программа

Теперь рассмотрим что-то совершенно иное: в примере 1.4 представлена программа, которая выполняет более сложную серию задач. Не рассчитывайте, что сразу поймете, как она работает, — книга для того и предназначена, чтоб научить вас этому! Таким образом я даю вам возможность увидеть и прочувствовать типичную полноразмерную программу, написанную на языке Python. Если вы знаете другие языки программирования, то можете сравнить их с Python прямо сейчас. Сможете ли вы, не зная Python и еще не прочтя расшифровку, примерно представить, что делает каждая строка? Вы уже видели примеры использования списка и словаря, а эта программа демонстрирует еще несколько новых возможностей.

В первом издании книги программа из примера подключалась к сайту YouTube и получала информацию о самых популярных роликах, таких как *Charlie Bit My Finger*. Она хорошо работала до того момента, как компания Google отключила поддержку этой службы. Во втором издании уже в новом примере (пример 1.4) мы подключаемся к другому сайту, который, очевидно, просуществует гораздо дольше, — *Wayback Machine* из Internet Archive (http://archive.org/) (бесплатного сервиса, сохраняющего миллиарды веб-страниц, в том числе фильмы, телешоу, музыкальные композиции, игры и иные цифровые артефакты за последние 20 лет). Еще несколько примеров таких *веб-API* вы увидите в главе 18.

Программа попросит вас ввести URL и дату. Затем она спросит у Wayback Machine, имеется ли копия этого веб-сайта за указанную дату. Если копия есть, API вернет информацию о ней программе, которая, в свою очередь, выведет URL и отобразит его в веб-браузере. Суть заключается в том, чтобы увидеть, как Python справляется с разнообразными задачами — принимает пользовательские данные, общается с веб-сайтами в Интернете и получает от них данные, извлекает оттуда URL и убеждает веб-браузер отобразить этот URL.

Если бы мы получали обычную веб-страницу, заполненную текстом, отформатированным как HTML, нам пришлось бы сначала придумать, как отобразить ее, а потом выполнить много действий — все это можно радостно перепоручить веб-браузеру. Мы также можем попробовать извлечь именно те данные, которые нам нужны (более подробно о *веб-скрапинге* читайте в главе 18). Любой из выбранных вариантов потребует выполнения большего количества работы и увеличит программу. Вместо этого Wayback Machine возвращает данные в формате JSON. JSON, или JavaScript Object Notation, — это читабельный для человека текстовый формат, который описывает типы и значения, а также выстраивает данные в определенном порядке. Он немного похож на языки программирования и уже стал популярным способом обмена данными между разными языками программирования и системами. Подробнее о JSON вы узнаете в главе 12.

Программы, написанные на языке Python, могут преобразовывать текст формата JSON в *структуры данных* (с которыми вы познакомитесь в следующих нескольких главах), как если бы вы написали программу для их создания самостоятельно. Наша небольшая программа выбирает лишь один фрагмент данных (URL старой веб-страницы, хранящейся в архиве). И опять же это полноценная программа, которую вы можете запустить самостоятельно. Мы почти не проверяли данные на ошибки, чтобы пример оставался коротким. Номера строк не являются частью программы и включены только для того, чтобы вам было проще следовать описанию, представленному после кода.

Пример 1.4. archive.py

```
1 import webbrowser
2 import json
3 from urllib.request import urlopen
4
5 print("Let's find an old website.")
6 site = input("Type a website URL: ")
7 era = input("Type a year, month, and day, like 20150613: ")
8 url = "http://archive.org/wayback/available?url=%s&timestamp=%s" % (site, era)
```

```
9 response = urlopen(url)
10 contents = response.read()
11 text = contents.decode("utf-8")
12 data = json.loads(text)
13 try:
14     old_site = data["archived_snapshots"]["closest"]["url"]
15     print("Found this copy: ", old_site)
16     print("It should appear in your browser now.")
17     webbrowser.open(old_site)
18 except:
19     print("Sorry, no luck finding", site)
```

Такая небольшая программа, написанная на языке Python, делает многое с помощью всего нескольких строк. Не все термины вы уже знаете, однако сможете познакомиться с ними в следующих главах.

1. *Импортируем* (делаем доступным для этой программы) весь код из модуля *стандартной библиотеки*, который называется `webbrowser`.
2. Импортируем весь код из модуля стандартной библиотеки, который называется `json`.
3. Импортируем только *функцию* `urlopen` из модуля стандартной библиотеки `urllib.request`.
4. Пустая строка (мы не хотим перегрузить восприятие).
5. Выводим на экран приветственный текст.
6. Выводим на экран вопрос об URL, считываем пользовательский ввод и сохраняем это в *переменной* с именем `site`.
7. Выводим на экран еще один вопрос и на этот раз считываем год, месяц и день, а затем сохраняем их в переменной с именем `era`.
8. Создаем строковую переменную с именем `url`, чтобы сайт Wayback Machine искал копию требуемого сайта по дате.
9. Соединяемся с сервером, расположенным по этому адресу, и запрашиваем определенный *веб-сервис*.
10. Получаем ответ и присваиваем его переменной `contents`.
11. *Дешифруем* содержимое переменной `contents` в текстовую строку формата JSON и приписываем ее переменной `text`.
12. Преобразуем переменную `text` в `data` — структуру данных языка Python, предназначенную для работы с видео.
13. Проверяем на ошибки: помещаем следующие четыре строки в блок `try` и, если находим ошибку, запускаем последнюю строку программы (она идет после ключевого слова `except`).
14. Получив совпадение по сайту и дате, извлекаем нужное значение из трехуровневого *словаря* Python. Обратите внимание на то, что в этой и двух последующих строках используются отступы — тем самым Python легче понять, что данные строки находятся в блоке `try`.

15. Выводим на экран полученный URL.
16. Сообщаем о том, что случится, когда выполнится следующая строка.
17. Отображаем полученный URL в браузере.
18. Если во время выполнения предыдущих строк что-то пошло не так, Python перейдет сюда.
19. Если программа дала сбой, выводим сообщение и имя сайта, который мы искали. Эта строка также имеет отступ, поскольку должна выполняться только в том случае, если выполняется строка `except`.

Когда я сам запустил эту программу в окне терминала, то ввел URL сайта и дату и получил следующий результат:

```
$ python archive.py
Let's find an old website.
Type a website URL: lolcats.com
Type a year, month, and day, like 20150613: 20151022
Found this copy: http://web.archive.org/web/20151102055938/http://www.lolcats.com/
It should appear in your browser now.
```

На рис. 1.3 показано то, что появилось в моем браузере.

Рис. 1.3. Результат обращения к Wayback Machine

В предыдущем примере мы задействовали стандартные библиотечные модули (программы, включаемые в Python при установке), но совсем не обязательно

ограничиваться только ими: на языке Python написано много отличного стороннего ПО. В примере 1.5 показывается та же программа, получающая доступ к архиву Интернета (Internet Archive), но в ней использован внешний пакет ПО для Python, который называется `requests`.

Пример 1.5. archive2.py

```
1 import webbrowser
2 import requests
3
4 print("Let's find an old website.")
5 site = input("Type a website URL: ")
6 era = input("Type a year, month, and day, like 20150613: ")
7 url = "http://archive.org/wayback/available?url=%s&timestamp=%s" % (site, era)
8 response = requests.get(url)
9 data = response.json()
10 try:
11     old_site = data["archived_snapshots"]["closest"]["url"]
12     print("Found this copy: ", old_site)
13     print("It should appear in your browser now.")
14     webbrowser.open(old_site)
15 except:
16     print("Sorry, no luck finding", site)
```

Новая версия короче и, как мне кажется, более читабельна для большинства людей. О `requests` вы узнаете в главе 18, а о других авторских программах для Python в главе 11.

Python в реальном мире

Стоит ли тратить время и силы на изучение Python? Язык программирования Python существует примерно с 1991 года (он старше Java, но моложе C) и является одним из пяти самых популярных языков программирования. Людям платят деньги за написание программ на Python — очень важных и значимых, которыми мы пользуемся каждый день: Google, YouTube, Instagram, Netflix и Hulu.

Я использовал Python для создания приложений в самых разных областях. Python имеет репутацию высокопроизводительного языка программирования, и это особенно нравится динамично развивающимся компаниям.

Python используется во многих компьютерных приложениях, таких как:

- командная строка на мониторе или в окне терминала;
- пользовательские интерфейсы (Graphical User Interface, GUI), включая сетевые;
- веб-приложения, как клиентские, так и серверные;
- бэкенд-серверы, поддерживающие крупные популярные сайты;
- *облака* (серверы, управляемые сторонними организациями);
- приложения для мобильных устройств;
- приложения для встроенных устройств.

Программы, написанные на Python, могут быть как одноразовыми *сценариями* — вы видели их ранее в этой главе, так и сложными системами, содержащими миллионы строк.

В опросе The 2018 Python Developers' Survey (https://www.jetbrains.com/research/python-developers-survey-2018/) вы можете увидеть числа и графики, показывающие текущее место языка Python в мире вычислительных машин.

Мы рассмотрим применение Python для создания сайтов, системного администрирования и манипулирования данными. Рассмотрим также использование Python в искусстве, науке и бизнесе.

Python против языка с планеты X

Насколько Python хорош по сравнению с другими языками программирования? Где и когда следует использовать тот или иной язык? В этом разделе я покажу примеры кода, написанные на других языках, чтобы вы могли оценить, с чем конкурирует Python. Вы *не* обязаны понимать каждый из приведенных фрагментов, если не работали с этими языками. (А когда увидите последний фрагмент, написанный на Python, то почувствуете облегчение из-за того, что не работали с некоторыми другими языками.) Если же вам интересен только Python — вы ничего не потеряете, если не станете читать этот раздел.

Каждая программа должна напечатать 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 (https://ru.wikipedia.org/wiki/Cmd.exe), она выполняет *пакетные* файлы, имеющие расширение `.bat`. Для Linux и других операционных систем семейства Unix (включая macOS) существует множество программ-оболочек. Самая популярная из них называется bash (https://www.gnu.org/software/bash/) или sh. Оболочка обладает простейшими 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. Подобные программы могли быть в числе самых первых в вашей карьере программиста. Проблема в том, что возможности для масштабирования у сценариев оболочки ограничиваются несколькими сотнями 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 `test.sh` и запустите его с помощью команды `sh test.sh`, то на экране увидите следующее:

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

Старые добрые C (https://ru.wikipedia.org/wiki/Си_(язык_программирования)) и C++ (https://ru.wikipedia.org/wiki/C++) являются довольно низкоуровневыми языками программирования, которыми пользуются в том случае, когда важна скорость. Ваша операционная система и множество других программ (включая программу python на вашем компьютере), скорее всего, написаны на C и C++.

Эти языки программирования труднее выучить и труднее поддерживать знания в актуальном состоянии. Вам придется отслеживать множество деталей, таких как *управление памятью*, 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! See? Si!\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 my little brother!" << \
        endl;
    return(0);
}
```

Java (https://www.java.com/ru/) и C# (https://docs.microsoft.com/en-us/dotnet/csharp/) являются преемниками языков C и C++. Они избавлены от некоторых недостатков предшественников (особенно в управлении памятью), но при этом могут быть немного избыточными. Следующий пример написан на Java:

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

Если вы никогда не писали ни на одном из этих языков, у вас может возникнуть вопрос, *что все это такое*? Ведь мы хотели всего лишь вывести на экран простую строку — действительно, некоторые языки нагружены значительным синтаксическим багажом. Подробнее об этом вы узнаете из главы 2.

C, C++ и Java являются примерами *статических языков*. Они требуют, чтобы вы указали компьютеру некоторые низкоуровневые детали, например типы данных. В приложении А говорится, что для разных типов данных выделяется разное количество памяти и для них можно выполнять лишь заранее определенный набор операций. *Динамические языки* (они также называются *скриптовыми*) — полная

противоположность статическим, они не заставляют вас определять тип переменной перед тем, как ее использовать.

Многоцелевым динамическим языком многие годы был Perl (https://www.perl.org/), очень мощный и с обширными библиотеками. Однако его синтаксис достаточно труден для понимания, а сам язык теряет в популярности из-за появления языков программирования Python и Ruby. Следующий пример побалует вас острым привкусом Perl:

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

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

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

Язык программирования PHP (http://www.php.net/) из следующего примера очень популярен в области веб-разработок, поскольку позволяет довольно легко объединять HTML и код. Однако язык PHP имеет несколько подводных камней, и с ним довольно трудно работать за пределами сферы веб-разработок. Вот так выглядит программа, написанная на PHP:

```
<?PHP
$language = 6;
echo "Language $language: I am PHP, a language and palindrome.\n";
?>
```

Язык Go (https://golang.org/) (в поисковике лучше искать *Golang*) появился относительно недавно и пытается быть эффективным и быстрым:

```
package main

import "fmt"

func main() {
  language := 7
  fmt.Printf("Language %d: Hey, ho, let's Go!\n", language)
}
```

Еще одной современной альтернативой языкам C и C++ является Rust (https://www.rust-lang.org/learn):

```
fn main() {
    println!("Language {}: Rust here!", 8)
```

Кто остался? Ах да, Python (https://www.python.org/):

```
language = 9
print(f"Language {language}: I am Python. What's for supper?")
```

Почему же Python?

Одной из причин, не обязательно самой важной, является популярность. Есть такие факты:

- Python — это наиболее быстро набирающий популярность основной язык программирования, как показано на рис. 1.4 (https://stackoverflow.blog/2017/09/06/incredible-growth-python/);
- редакторы TIOBE Index (https://www.tiobe.com/tiobe-index/) в июне 2019 года заявили: «В этом месяце Python снова достиг высочайшей позиции в TIOBE Index, его результат составил 8,5 %. Если Python удержит такой темп, то в течение ближайших трех-четырех лет он сможет заменить C и Java, став самым популярным языком программирования в мире»;
- Python стал языком программирования 2018 года по версии TIOBE и занял первые места в рейтингах IEEE Spectrum (https://spectrum.ieee.org/at-work/innovation/the-2018-top-programming-languages) и PyPL (http://pypl.github.io/PYPL.html);
- Python является самым популярным языком программирования для курсов введения в информатику в лучших американских колледжах (https://cacm.acm.org/blogs/blog-cacm/176450-python-is-now-the-most-popular-introductory-teaching-language-at-top-u-s-universities/fulltext);
- Python официально используется для обучения во французских гимназиях.

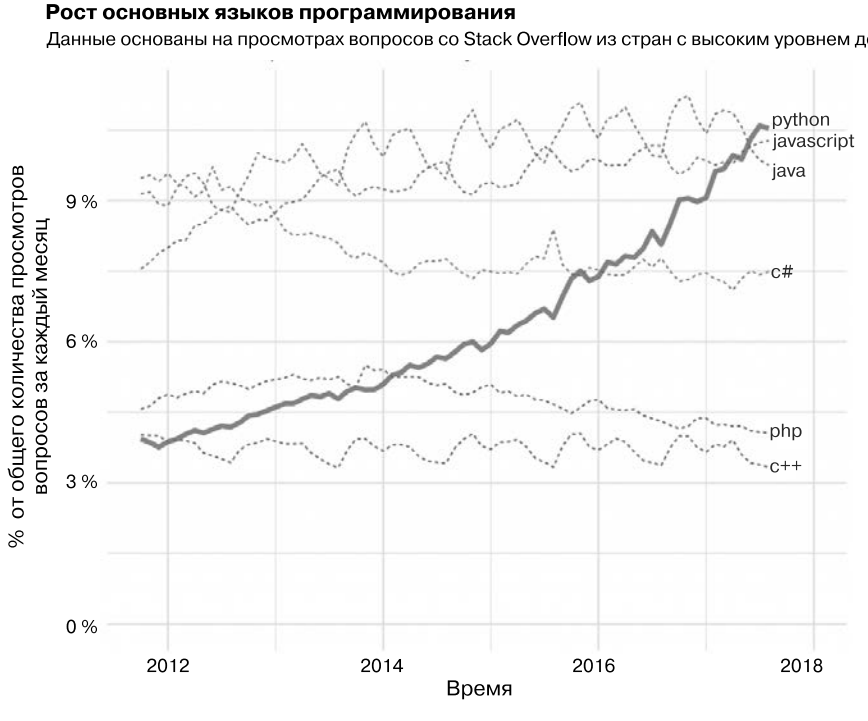

Рис. 1.4. Python опережает в росте остальные основные языки программирования

В последнее время Python стал чрезвычайно популярным языком программирования в науке о данных и машинном обучении. Если вы хотите получить высокооплачиваемую работу разработчика в интересной области, Python — хороший выбор. А если вы занимаетесь набором персонала, имейте в виду, что пул опытных разработчиков на Python постоянно растет.

Но *почему* Python так популярен? Ведь языки программирования не имеют харизмы. В чем же причина?

Python — многоцелевой высокоуровневый язык программирования. Его дизайн позволяет писать хорошо *читаемый* код, что на самом деле гораздо важнее, чем кажется на первый взгляд. Каждая компьютерная программа пишется всего однажды, но впоследствии к ней обращаются множество раз. Благодаря удобочитаемости программу легко запомнить, а также легко ее *воспроизвести*. По сравнению с другими популярными языками программирования кривая обучения языку Python более гладкая, что позволяет ученику быстрее стать продуктивным. Однако есть и сложные моменты, с которыми вы столкнетесь по мере приобретения опыта.

Относительный лаконизм языка Python позволяет создавать гораздо более короткие программы — аналогичная программа, но написанная на статическом языке, будет намного длиннее. 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.

С помощью Python вы можете написать все, что хотите, и совершенно бесплатно использовать это где угодно. Никто не скажет, прочитав вашу программу: «Ах, какая милая программка! Будет жаль, если с ней что-нибудь случится».

Python запускается практически везде и имеет «встроенные батарейки» — огромное количество разнообразного ПО в стандартных библиотеках. В этой книге имеется множество примеров использования стандартной библиотеки и полезного стороннего кода.

Но основная причина использования Python вам, возможно, покажется неожиданной: как правило, люди *любят* работать с этим языком, а не рассматривают его как необходимое зло для решения задачи — он подходит их образу мышления. Часто разработчики, когда им приходится программировать на другом языке, говорят, что им не хватает какой-то возможности Python. И это выделяет Python на фоне всех его «коллег».

Когда не стоит использовать Python

Python не всегда будет наилучшим выбором.

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

Python достаточно быстрый для большинства приложений, но его скорости может оказаться недостаточно для наиболее требовательных из них. Если ваша программа проводит большую часть времени за вычислениями, что в технических

терминах называется *«ограничена быстродействием процессора (CPU-bound)»*, то языки C, C++, Java, Rust или Go справятся с задачей гораздо лучше, чем Python. Но не всегда!

Учтите следующие обстоятельства.

- Иногда более качественный *алгоритм* (пошаговое решение) в Python превосходит неэффективный алгоритм в C. Более высокая скорость разработки в Python дает больше времени для экспериментов в поисках альтернативных решений.
- Во многих приложениях (особенно в веб-приложениях) программа «бьет баклуши» 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 главе 19.
- Интерпретаторы Python становятся быстрее. Java, когда только появился, был чрезвычайно медленным, и на его ускорение ушло много времени и денег. Языком программирования Python не владеет ни одна корпорация, поэтому он улучшается последовательно и более плавно. В подразделе «PyPy» на с. 465 я расскажу о проекте *PyPy* и его приложениях.
- У вас в работе может быть очень сложное приложение, и тогда, независимо от того, что вы делаете, Python не сможет удовлетворить все ваши требования. Обычной альтернативой в таком случае являются языки программирования C, C++ и Java. Вы также можете рассмотреть возможность использования языка Go (http://golang.org/), который выглядит как Python, но работает как C, или языка Rust.

Python 2 против Python 3

Вы можете столкнуться с проблемой выбора одной из двух версий Python. Python 2 существует давно и предустановлен на компьютерах с Linux и Apple. Это был отличный язык, но нет ничего идеального. В языках программирования, как и во многих других сферах, бывают ошибки косметические и легко исправимые, а бывают — сложные и требующие усилий. Исправления *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 (https://www.python.org/~guido) и другие разработчики решили собрать и объединить все исправления и в результате в 2008 году представили миру Python 3. Python 2 — это прошлое, а Python 3 — будущее. Финальная версия Python 2 имеет номер 2.7, и некоторое время она еще будет поддерживаться, однако на ней род заканчивается; Python 2.8 никогда не выйдет. Окончание поддержки языка Python 2 намечено на январь 2020 года. Больше не будут исправляться проблемы (в том числе проблемы с безопасностью), и многие весомые пакеты Python к этому моменту перестанут поддерживать Python 2 (https://python3statement.org/). В операционных системах также будет

отключен Python 2 и, скорее всего, подключен Python 3 в качестве нового языка, используемого по умолчанию. Преобразование популярного ПО в Python 3 было постепенным, но переломный момент уже произошел и новые разработки будут вестись на Python 3.

Эта книга посвящена Python 3. Он выглядит практически так же, как и Python 2. Самое очевидное изменение — это тот факт, что `print` в Python 3 является функцией, поэтому вам нужно вызывать ее с помощью круглых скобок, в которых будут перечислены аргументы. А самое главное изменение — это обработка символов *Unicode* (она рассматривается в главе 12). На протяжении всей книги я буду обращать ваше внимание и на другие различия.

Установка Python

Чтобы не загромождать текст, я вынес детали установки Python 3 в приложение Б. Если у вас еще не установлен Python 3 или вы не до конца в этом уверены, обратитесь к приложению и посмотрите, каковы должны быть ваши действия. Да, это может быть хлопотно и трудоемко, но сделать это вам придется лишь однажды.

Запуск Python

После установки рабочей копии Python 3 вы сможете использовать ее, чтобы запускать как приведенные в этой книге программы, так и собственный код. Как же запустить программу, написанную на языке Python? Существует два основных способа.

- Встроенный в Python *интерактивный интерпретатор* (также он называется *оболочкой*) предоставляет простой способ поэкспериментировать с небольшими программами. Строка за строкой вы вводите команды и мгновенно видите результат — такая тесная связь между набором текста и его просмотром позволяет проводить эксперименты быстрее. Я буду использовать интерактивный интерпретатор для демонстрации возможностей языка, а вы те же команды можете вводить в собственном компьютере.
- В остальных случаях 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 встроенный интерактивный интерпретатор. Когда вы вводите команду из примера и получаете тот же результаты, вы понимаете, что находитесь на правильном 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 запуска вам следует ввести именно ее имя.

Интерактивный интерпретатор работает практически так же, как и интерпретатор для файлов, за одним исключением: когда вы вводите обычное значение, интерактивный интерпретатор автоматически выводит его на экран. Это не часть языка Python, а всего лишь особенность интерпретатора, которая позволяет вам сэкономить немного времени, не набирая конструкцию `print()` каждый раз. Например, если вы запустите Python и введете в интерпретатор число `27`, оно будет продублировано в терминале (если в вашем файле есть строка `27`, Python не расстроится, но при запуске программы вы не увидите ничего):

```
$ python
Python 3.7.2 (v3.7.2:9a3ffc0492, Dec 24 2018, 02:44:43)
[Clang 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> 27
27
```

В предыдущем 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.

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

```
>>> print(27)
27
```

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

```
print(27)
```

Создадим файл программы Python и запустим его.

1. Откройте текстовый редактор.
2. Введите в него строку `print(27)`, как это показано здесь.
3. Сохраните этот файл с именем `test.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 test.py
```

Вы должны увидеть такую строку:

```
27
```

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

Что дальше?

Вы будете вводить команды в работающую систему Python, и они должны соответствовать синтаксису языка. Вместо того чтобы сваливать на вас все синтаксические правила сразу, мы неторопливо пройдемся по ним в нескольких следующих главах.

Базовый 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 предоставить вам графические пользовательские интерфейсы, помогающие в редактировании текста, и экраны помощи. Более подробно об этом вы прочитаете в главе 19.

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 подобную «пасхалку»). 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.
Читаемость имеет значение.
Особые случаи не настолько особые, чтобы нарушать правила.
```

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

```
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 поначалу он может быть и неочевиден, 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 всей книги я буду приводить примеры, иллюстрирующие эти утверждения.

Читайте далее

В следующей главе мы поговорим о типах данных и переменных в Python. Это подготовит вас к чтению тех глав, в которых подробно рассматриваются типы данных и структуры кода Python.

Упражнения

Эта глава была введением в язык программирования Python. Вы узнали, что язык делает, как выглядит и где его можно применить. В конце каждой главы я буду предлагать выполнить небольшие задания, которые помогут вам запомнить то, что вы только что прочитали, и подготовят к следующим урокам.

1.1. Если вы еще не установили Python 3, сделайте это сейчас. Прочтите приложение Б, чтобы узнать детали.

1.2. Запустите интерактивный интерпретатор Python 3. Детали опять же вы найдете в приложении Б. Интерпретатор должен вывести несколько строк о себе, а затем строку, начинающуюся с символов `>>>`. Перед вами приглашение для ввода команд Python.

1.3. Немного поэкспериментируйте с интерпретатором. Используйте его как калькулятор и наберите `8 * 9`. Нажмите клавишу Enter, чтобы увидеть результат. Python должен вывести `72`.

1.4. Теперь введите число `47` и нажмите клавишу Enter. Появилось ли число `47` в следующей строке?

1.5. Теперь введите `print(47)` и нажмите клавишу Enter. Появилось ли снова число `47` в следующей строке?

ГЛАВА 2

Данные: типы, значения, переменные и имена

Доброе имя лучше большого богатства.

Притчи 22:1

Все, что хранится в компьютере, представляет собой лишь последовательность *битов* (приложение А). Одно из преимуществ вычислительной техники заключается в том, что мы можем интерпретировать эти биты любым удобным нам способом — как данные всевозможного размера и типов (например, как числа или текстовые символы) или даже как компьютерный код. Мы используем Python для определения наборов этих битов, соответствующих разным задачам, а также для отправки их в процессор и получения обратно.

Мы начнем с *типов данных*, используемых в Python, и *значений*, которые они могут содержать. Затем рассмотрим, как представить данные в виде значений-*литералов* и *переменных*.

В Python данные являются объектами

Память вашего компьютера визуально можно представить в виде длинного ряда полок. Каждый слот на этих полках имеет ширину 1 байт (8 бит). Слоты пронумерованы от 0 (первая позиция) до самого конца. Современные компьютеры имеют миллиарды байт памяти (гигабайт), поэтому полки могли бы заполнить огромный воображаемый склад.

Программа Python получает доступ к определенной области памяти вашего компьютера с помощью операционной системы. Эта память используется для кода самой программы, а также для данных, которыми программа оперирует. Операционная система гарантирует, что программа не может читать или записывать в другие области памяти без соответствующего разрешения.

Программы следят за тем, *где* (область памяти) хранятся их биты и *чем* (тип данных) они являются. С точки зрения вашего компьютера все биты одинаковы.

Одни и те же биты могут иметь разные значения в зависимости от того, какого они типа. Один и тот же вариант расстановки битов может означать как число `65`, так и текстовый символ `A`.

Разные типы используют разное количество битов. Когда вы читаете о «64-битной машине», это означает, что целое число использует 64 бита (8 байт).

Некоторые языки хранят эти необработанные значения в памяти, отслеживая их размеры и типы. Вместо непосредственной обработки таких данных Python упаковывает каждое значение — булевы значения, целые числа, числа с плавающей точкой, строки и даже крупные структуры данных, функции и программы — в память как *объекты*. Глава 10 этой книги посвящена созданию собственных объектов в Python, но сейчас мы поговорим только об объектах, которые обрабатывают встроенные типы данных.

Продолжая аналогию с полками: можно представить, что объекты — это коробки переменной длины, занимающие место на этих полках так, как показано на рис. 2.1. Python создает такие коробки, размещает их на свободных местах и убирает, когда потребность в них отпадает.

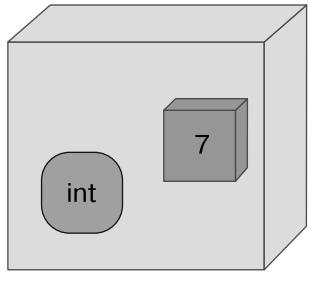

Рис. 2.1. Похожий на коробку объект — целое число со значением 7

В Python объектом является фрагмент данных, в котором содержится как минимум следующее:

- *тип*, определяющий, что объект может делать (подробнее см. в следующем разделе);
- уникальный *идентификатор*, позволяющий отличить его от других объектов;
- *значение*, соответствующее типу;
- счетчик *ссылок* для отслеживания того, как часто объект используется.

Идентификатор представляет собой адрес места на полке. *Тип* похож на фабричный штамп на коробке, который поясняет, что объект может делать. Если объект является целым числом, он имеет тип `int` и может (помимо всего прочего, о чем вы узнаете из главы 3) быть добавлен к другому объекту с типом `int`. Если мы представим, что коробка сделана из прозрачного пластика, то сможем увидеть *значение*, находящееся внутри нее. О том, зачем нужен *счетчик ссылок*, вы узнаете через несколько разделов, когда мы будем говорить о переменных и именах.

Типы

В табл. 2.1 представлены базовые типы данных в Python. Во втором столбце («Тип») содержится имя этого типа в Python. Третий столбец («Изменяемый?») указывает, можно ли изменить значение переменной после ее создания (подробнее об этом поговорим в следующем разделе). В столбце «Примеры» показываются один или

несколько примеров-литералов, соответствующих этому типу. И последний столбец («Глава») указывает на главу этой книги с наиболее подробной информацией о данном типе.

Таблица 2.1. Базовые типы данных в Python

Имя	Тип	Изменяемый?	Примеры	Глава
Булево значение	bool	Нет	True, False	Глава 3
Целое число	int	Нет	47, 25000, 25_000	Глава 3
Число с плавающей точкой	float	Нет	3.14, 2.7e5	Глава 3
Комплексное число	complex	Нет	3j, 5 + 9j	Глава 22
Текстовая строка	str	Нет	'alas', "alack", '''a verse attack'''	Глава 5
Список	list	Да	['Winken', 'Blinken', 'Nod']	Глава 7
Кортеж	tuple	Нет	(2, 4, 8)	Глава 7
Байты	bytes	Нет	b'ab\xff'	Глава 12
Массив байтов	bytearray	Да	bytearray(...)	Глава 12
Множество	set	Да	set([3, 5, 7])	Глава 8
Фиксированное множество	frozenset	Нет	frozenset(['Elsa', 'Otto'])	Глава 8
Словарь	dict	Да	{'game': 'bingo', 'dog': 'dingo', 'drummer': 'Ringo'}	Глава 8

После глав, посвященных этим базовым типам, в главе 10 вы узнаете, как создавать новые типы данных.

Изменчивость

Изменчивость одна лишь неизменна.

Перси Шелли

Тип также определяет, можно ли значение, которое хранится в ящике, изменить — тогда это будет *изменяемое значение*, или оно константно — *неизменяемое значение*. Неизменяемый объект как будто находится в закрытом ящике с прозрачными стенками (см. рис. 2.1): увидеть значение вы можете, но не в силах его изменить. По той же аналогии изменяемый объект похож на коробку с крышкой: вы можете не только увидеть хранящееся там значение, но и изменить его, не изменив его тип.

Python является *строго типизированным* языком, а это означает, что тип объекта не изменяется, даже если его значение изменяемо (рис. 2.2).

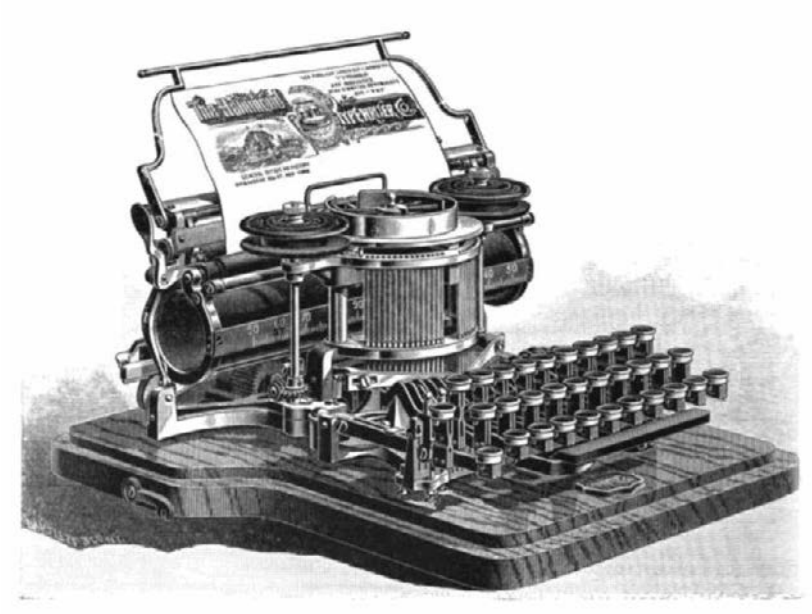

Рис. 2.2. Строгая типизация не означает, что клавиши нужно нажимать сильнее

Значения-литералы

Существует два вида определения данных в Python:

- как литералы;
- как переменные.

В следующих главах вы увидите, как указываются значения-литералы для разных типов данных — целые числа представляют собой последовательность цифр, дробные числа содержат десятичную точку, текстовые строки заключаются в кавычки и т. д. Но в примерах этой главы — чтобы избежать излишней сложности — мы будем использовать лишь короткие целые числа из десятичной системы счисления и один-два списка. Десятичные целые числа такие же, как числа в математике: они представляют собой последовательность цифр от 0 до 9. В главе 3 мы рассмотрим дополнительные детали работы с целыми числами (например, знаки и недесятичные системы счисления).

Переменные

Вот мы и добрались до ключевого понятия языков программирования.

Python, как и большинство других компьютерных языков, позволяет вам определять *переменные* — имена для значений в памяти вашего компьютера, которые вы далее будете использовать в программе.

Имена переменных в Python отвечают определенным правилам.

- Они 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 (_).
- Они *чувствительны к регистру*: `thing`, `Thing` и `THING` — это разные имена.
- Они должны начинаться с буквы или нижнего подчеркивания, но не с цифры.
- Python особо обрабатывает имена, которые начинаются с нижнего подчеркивания (об этом вы сможете прочитать в главе 9).
- Они не могут совпадать с *зарезервированными словами* Python (их также называют ключевыми).

Перед вами список зарезервированных[1] слов:

```
False      await      else       import     pass
None       break      except     in         raise
True       class      finally    is         return
and        continue   for        lambda     try
as         def        from       nonlocal   while
assert     del        global     not        with
async      elif       if         or         yield
```

Внутри программы Python увидеть список зарезервированных слов можно с помощью команд:

```
>>> help("keywords")
```

или:

```
>>> import keyword
>>> keyword.kwlist
```

Корректными являются такие имена:

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

А следующие имена некорректны:

- `1`;
- `1a`;
- `1_`;
- `name!`;
- `another-name`.

[1] async и await появились в Python 3.7.

Присваивание

В Python символ = применяется для присваивания значения переменной.

В школе нас 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, чем проверка на равенство.

Программы непохожи на алгебру. В школе мы имели дело с подобными уравнениями:

```
y = x + 12
```

Решить уравнение можно, подставив значение для x. Если вы зададите для x значение 5, то, поскольку 5 + 12 равно 17, значение y будет равно 17. Подставьте значение 6, и y будет равен 18. И так далее.

Строки компьютерной программы могут выглядеть как уравнения, но означают они при этом нечто иное. В Python и других компьютерных языках x и y являются переменными. Python знает, что цифра или простая последовательность цифр вроде 12 или 5 является числовым литералом. Рассмотрим небольшую программу на Python, которая схожа с этим уравнением, — она выводит на экран значение y:

```
>>> x = 5
>>> y = x + 12
>>> y
17
```

Здесь мы видим большое различие между математикой и программами: в математике знак = означает равенство обеих сторон, а в программировании он означает *присваивание: переменной слева мы присваиваем значение с правой стороны*.

В программировании также принято, что все находящееся справа от знака = должно иметь значение (это называется *инициализацией*). Справа вы можете увидеть значение-литерал, переменную, которой было присвоено значение, или их комбинацию. Python знает, что 5 и 12 — это числовые литералы. В первой строке целочисленное значение 5 присваивается переменной x. Теперь мы можем использовать переменную x в следующей строке. Когда Python читает выражение y = x + 12, он делает следующее:

- видит знак = в середине;
- понимает, что это оператор присваивания;
- вычисляет значение с правой стороны (получает значение объекта, на который ссылается переменная x, и добавляет его к 12);
- присваивает этот результат переменной слева — y.

Теперь, введя имя y в интерактивном интерпретаторе, можно увидеть его новое значение.

Если вы начнете программу со строки y = x + 12, Python сгенерирует исключение (ошибку), поскольку переменная x еще не имеет значения:

```
>>> y = x + 12
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'x' is not defined
```

Более подробно об исключениях можно прочитать в главе 9. На компьютерном языке мы скажем, что переменная x *не была инициализирована*.

В алгебре вы могли бы сделать все наоборот — присвоить значение y, чтобы подсчитать значение x. Для того, чтобы сделать это в Python, вам нужно получить значения-литералы и инициализированные переменные с правой стороны оператора присваивания до того, как присвоить значение переменной x:

```
>>> y = 5
>>> x = 12 - y
>>> x
7
```

Переменные — это имена, а не локации

Пришло время сделать важное утверждение о переменных в Python: переменные — *всего лишь имена*, и в этом заключается отличие Python от других языков программирования. Об этом важно помнить, особенно при работе с такими *изменяемыми* объектами, как списки. Операция присваивания *не копирует* значение, а только лишь *прикрепляет имя* к объекту, содержащему нужные данные. Имя — это *ссылка* на объект, а не сам объект. Можно представить, что имя — это этикетка, приклеенная на коробку с объектом, которая размещается где-то в памяти компьютера (рис. 2.3).

В других языках программирования переменные сами по себе имеют тип и привязываются к локации в памяти. Вы можете изменить значение в этой локации, но оно должно быть того же типа. Именно поэтому в статических языках нужно объявлять тип переменных. В Python этого делать не требуется, поскольку имя может ссылаться на все что угодно: значение и тип мы получаем, идя по цепочке к самому объекту с данными. Такой подход экономит время, но при этом имеет свои недостатки.

- Вы можете неверно написать имя переменной и получить исключение, поскольку она ни на что не ссылается, Python не выполняет такую проверку автоматически в отличие от статических языков. В главе 19 показывается способ предварительной проверки переменных.
- В сравнении с такими языками, как C, у Python скорость работы ниже. Ведь он заставляет компьютер выполнять больше работы, для того чтобы вам не пришлось выполнять ее самостоятельно.

Попробуйте сделать следующее с помощью интерактивного интерпретатора (рис. 2.4).

1. Как и раньше, присвойте значение `7` имени `a`. Это создаст объект-«ящик», содержащий целочисленное значение `7`.
2. Выведите на экран `a`.
3. Присвойте имя `a` переменной `b`, заставив `b` прикрепиться к объекту-«ящику», содержащему значение `7`.
4. Выведите `b`.

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

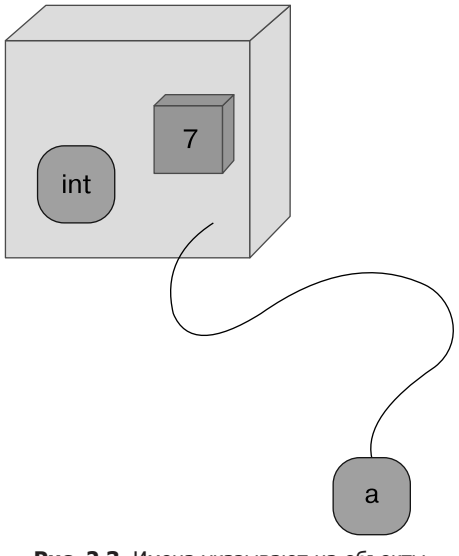

Рис. 2.3. Имена указывают на объекты (переменная указывает на целочисленный объект со значением 7)

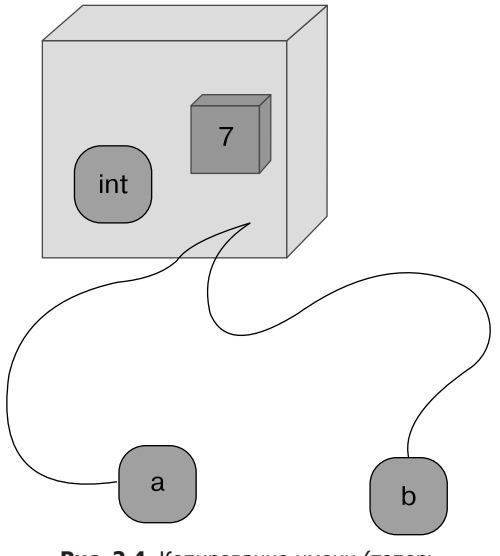

Рис. 2.4. Копирование имени (теперь переменная b указывает на тот же целочисленный объект)

В Python, если нужно узнать тип какого-либо объекта (переменной или значения), можно использовать конструкцию `type(`*`объект`*`)`. `type()` — одна из встроенных в Python функций. Чтобы проверить, указывает ли переменная на объект определенного типа, используйте конструкцию `isinstance(`*`type`*`)`:

```
>>> type(7)
<class 'int'>
>>> type(7) == int
True
>>> isinstance(7, int)
True
```

Когда я упоминаю функцию, то после ее имени размещаю круглые скобки (()) и таким образом подчеркиваю, что это именно функция, а не имя переменной или что-либо еще.

Попробуем проделать это с разными значениями (`58`, `99.9`, `'abc'`) и переменными (`a`, `b`):

```
>>> a = 7
>>> b = a
>>> 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 10). В Python значения терминов «класс» и «тип» примерно одинаковы.

Как вы могли заметить, при упоминании имени переменной Python ищет объект, на который она ссылается. Неявно Python выполняет большое количество действий и часто создает временные объекты, которые будут удалены спустя одну-две строки.

Снова рассмотрим пример, показанный ранее:

```
>>> y = 5
>>> x = 12 - y
>>> x
7
```

В этом фрагменте кода Python сделал следующее:

- создал целочисленный объект со значением `5`;
- создал переменную `y`, которая указывает на этот объект;
- нарастил счетчик ссылок для объекта, содержащего значение `5`;
- создал еще один целочисленный объект со значением `12`;
- вычел значение объекта, на который указывает переменная `y` (`5`), из значения `12`, содержащегося в анонимном объекте;
- присвоил результат (`7`) новому (пока еще безымянному) целочисленному объекту;
- заставил переменную `x` указывать на этот новый объект;
- нарастил счетчик ссылок для объекта, на который указывает переменная `x`;
- нашел значение объекта, на который ссылается переменная `x` (`7`), и вывел его на экран.

Когда количество ссылок на объект становится равным нулю, это означает, что ни одно имя на него больше не ссылается, поэтому хранить такой объект нет

необходимости. В Python имеется *сборщик мусора*, который позволяет повторно использовать память, занятую уже ненужными на данный момент объектами: представьте себе, будто кто-то следит за этими полками с памятью и забирает ненужные коробки на переработку.

В нашем случае объекты со значениями `5`, `12` и `7`, а также переменные `x` и `y` больше не нужны. Сборщик мусора Python может или отправить их в небесный рай для объектов[1], или сохранить, исходя из соображений производительности, так как небольшие целые числа используются довольно часто.

Присваивание нескольким именам

Вы можете присвоить значение сразу нескольким переменным одновременно:

```
>>> two = deux = zwei = 2
>>> two
2
>>> deux
2
>>> zwei
2
```

Переназначение имени

Поскольку имена указывают на объекты, если изменить значение, присвоенное имени, оно начнет указывать на другой объект. Счетчик ссылок старого объекта уменьшится на 1, а счетчик ссылок нового увеличится на ту же величину.

Копирование

Как вы видели на рис. 2.4, присваивание существующей переменной `a` новой переменной `b` заставит `b` указывать на тот же объект, что и `a`. Если вы выберете этикетку `a` или `b` и обратитесь к объекту, на который они указывают, вы получите одинаковый результат.

Если объект неизменяем (например, целое число), его значение нельзя изменить, поэтому по умолчанию оба имени являются доступными только для чтения. Попробуйте выполнить следующий код:

```
>>> x = 5
>>> x
5
>>> y = x
>>> y
5
>>> x = 29
>>> x
```

[1] Или на Остров забытых объектов.

```
29
>>> y
5
```

Когда мы присваиваем переменную `x` переменной `y`, переменная `y` начинает указывать на целочисленный объект со значением `5`, на который также указывает и переменная `x`. Далее мы изменяем переменную `x` так, чтобы она указывала на целочисленный объект со значением `29`. Объект со значением `5`, на который все еще указывает переменная `y`, не изменился.

В случае, когда оба имени указывают на *изменяемый* объект, вы можете изменить значение объекта с помощью любого имени. Если вы этого еще не знали, такая особенность может вас удивить.

Список представляет собой изменяемый массив значений (в главе 7 этот тип данных описывается более подробно). В нашем примере `a` и `b` указывают на список, содержащий три целочисленных объекта:

```
>>> a = [2, 4, 6]
>>> b = a
>>> a
[2, 4, 6]
>>> b
[2, 4, 6]
```

Эти элементы списка (`a[0]`, `a[1]` и `a[2]`) сами по себе являются именами, указывающими на целочисленные объекты со значениями `2`, `4` и `6`. Список хранит элементы в заданном порядке.

Теперь давайте изменим первый элемент списка с помощью имени `a` и убедимся, что список `b` также изменился:

```
>>> a[0] = 99
>>> a
[99, 4, 6]
>>> b
[99, 4, 6]
```

Когда первый элемент списка изменяется, он больше не указывает на объект со значением `2`. Теперь он указывает на объект со значением `99`. Список все еще имеет тип `list`, но его значения (элементы списка и их порядок) можно изменить.

Выбираем хорошее имя переменной

> Он говорил правильные вещи, но называл их
> неверными именами.
>
> *Элизабет Барретт Браунинг*

Удивительно, но выбор соответствующих имен для переменных очень важен. Во многих примерах кода, которые мы успели рассмотреть, я использовал простейшие имена вроде `a` и `x`. В реальных программах вам будет нужно отслеживать

гораздо больше переменных одновременно и придется балансировать между краткостью и понятностью. Например, имя `num_loons` можно напечатать быстрее, чем `number_of_loons` или `gaviidae_inventory`, однако они более понятны, чем имя `n`.

Читайте далее

Числа! Они такие увлекательные! Хотя вы, наверное, даже не предполагали насколько[1]. Вы увидите, как использовать Python в качестве калькулятора, а также узнаете, как кот помог создать систему счисления.

Упражнения

2.1. Присвойте целочисленное значение `99` переменной `prince` и выведите ее на экран.

2.2. Какого типа значение `5`?

2.3. Какого типа значение `2.0`?

2.4. Какого типа выражение `5 + 2.0`?

[1] Цифра 8 похожа на снеговика!

ГЛАВА 3
Числа

Поступай так, чтобы принести счастье многим людям.

Фрэнсис Хатчесон

В этой главе мы начнем рассматривать простейшие типы данных, используемые в Python:

- *булевы значения* (они могут быть равны `True` или `False`);
- *целые числа* (например, `42` или `100000000`);
- *числа с плавающей точкой* (числа с десятичной точкой вроде `3.14159`, а иногда и экспоненты вроде `1.0e8`, что означает *десять в восьмой степени*, или `100000000.0`).

В каком-то смысле они похожи на атомы. В этой главе мы будем использовать их по отдельности, а впоследствии вы узнаете, как объединить данные простейших типов в более крупные «молекулы», такие как списки и словари.

Для каждого типа определены свои правила использования и обработки компьютером. Я также покажу вам, как использовать литералы (например, `97` или `3.1416`) и *переменные*, которые упоминались в главе 2.

Примеры кода, показанные в этой главе, написаны корректно, но являются всего лишь фрагментами. Мы будем вводить эти фрагменты в интерактивный интерпретатор Python и сразу получать результат. Попробуйте запустить их самостоятельно на своем компьютере (такие примеры вы узнаете по подсказке `>>>`).

Булевы значения

В Python объекты булева (логического) типа данных могут иметь только два значения — `True` или `False`. Иногда вы будете использовать их напрямую, а иногда оценивать «истинность» других типов по их значениям. Специальная функция `bool()` может преобразовать любой тип данных Python в булев.

Функции мы рассмотрим в главе 9, сейчас же вам нужно знать только то, что функция имеет имя, от нуля и более входных *аргументов*, взятых в скобки

и разделенных запятыми, а также от нуля и более *возвращаемых значений*. Функция `bool()` в качестве аргумента принимает любое значение и возвращает его булев эквивалент.

Для ненулевых чисел эта функция вернет значение `True`:

```
>>> bool(True)
True
>>> bool(1)
True
>>> bool(45)
True
>>> bool(-45)
True
```

Нулевые значения преобразуются в значение `False`:

```
>>> bool(False)
False
>>> bool(0)
False
>>> bool(0.0)
False
```

Чем полезны булевы значения, вы узнаете в главе 4. В последующих главах мы поговорим о том, при каких обстоятельствах списки, словари и другие типы данных могут преобразовываться как `True` или `False`.

Целые числа

Целые числа — это целые числа без дробей и десятичной точки, ничего особенного. Кроме, возможно, знака и системы счисления (если вы хотите выразить числа в других системах, а не в десятичной).

Числа-литералы

Любая последовательность цифр в Python представляет собой *число-литерал*:

```
>>> 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
```

Исключение предупреждает вас о том, что вы ввели код, который нарушает правила Python. Что это значит, я объясню в подразделе «Системы счисления». В книге вы увидите еще много примеров исключений, поскольку они являются основным механизмом обработки ошибок в Python.

Запись целого числа вы можете начать с конструкций `0b`, `0o` или `0x`. См. подраздел «Системы счисления» далее в этой главе.

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
```

При записи целого числа нельзя использовать запятые:

```
>>> 1,000,000
(1, 0, 0)
```

Вместо числа «миллион» вы получите *кортеж* с тремя значениями (более подробная информация о кортежах находится в главе 7). Однако в качестве разделителя вы *можете* использовать символ «нижнее подчеркивание»[1].

```
>>> million = 1_000_000
>>> million
1000000
```

Нижние подчеркивания, которые будут просто проигнорированы, по вашему желанию ставятся на любую позицию после первой цифры:

```
>>> 1_2_3
123
```

Операции с целыми числами

На нескольких следующих страницах вы увидите примеры использования Python в качестве простого калькулятора. Можно выполнять обычные арифметические операции с помощью Python, используя математические *операторы* из этой таблицы.

Оператор	Описание	Пример	Результат
+	Сложение	5 + 8	13
–	Вычитание	90 – 10	80

Продолжение ⇩

[1] В версиях Python 3.6 и выше.

(Продолжение)

Оператор	Описание	Пример	Результат
*	Умножение	4 * 7	28
/	Деление с плавающей точкой	7 / 2	3.5
//	Целочисленное деление	7 // 2	3
%	Вычисление остатка	7 % 3	1
**	Возведение в степень	3 ** 4	81

Сложение и вычитание будут работать так, как вы и ожидаете:

```
>>> 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
```

Обратите внимание на то, что вставлять пробел между числом и оператором не обязательно:

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

Такой формат выглядит лучше стилистически и проще читается.

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* (они рассматриваются далее в этой главе):

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

Целочисленное деление вернет вам целочисленный ответ и отбросит остаток:

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

Вместо того чтобы проделать дыру в пространственно-временном континууме, 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
```

Целые числа и переменные

Во всех предыдущих примерах использовались непосредственно целочисленные значения, но можно смешивать целочисленные значения и переменные, которым было присвоено целочисленное значение:

```
>>> a = 95
>>> a
95
>>> a – 3
92
```

Как вы узнали во второй главе, `a` — это имя, которое указывает на целочисленный объект. 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
```

Опять же эта операция некорректна с точки зрения математики, но именно так вы выполняете повторное присваивание значения переменной в Python. 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 `a` значение временной переменной:

```
>>> 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 от знака =. Это гораздо быстрее и лучше на вид, чем использование временной переменной.

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

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

А это выражение аналогично выражению a = a + 8:

```
>>> a = 92
>>> a += 8
>>> a
100
```

Это — выражению a = a * 2:

```
>>> a = 100
>>> a *= 2
>>> a
200
```

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

```
>>> a = 200
>>> a /= 3
>>> a
66.66666666666667
```

Теперь попробуем использовать сокращенный вариант a = a // 4 (целочисленное деление):

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

Символ % в Python имеет несколько разных применений. 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 нечто новое — *функцию* с именем `divmod`: она получает целые числа `9` и `5`, а возвращает двухэлементный *кортеж*. Как я уже говорил, с кортежами вы познакомитесь в главе 7, а с функциями — в главе 9.

Рассмотрим последнюю математическую функцию, которая также позволяет работать одновременно с целыми числами и с числами с плавающей точкой — возведение в степень с помощью оператора **:

```
>>> 2**3
8
>>> 2.0 ** 3
8.0
>>> 2 ** 3.0
8.0
>>> 0 ** 3
0
```

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 более высокий *приоритет*, чем сложение, поэтому вы увидите ответ, совпадающий со второй версией:

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

Как узнать приоритет той или иной операции? В приложении Д есть большая таблица со всеми правилами. Однако на практике я и сам никогда в нее не смотрю, потому что гораздо проще добавить пару скобок, чтобы сгруппировать код и вычисления нужным образом:

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

Этот пример возведения в степень:

```
>>> -5 ** 2
-25
```

аналогичен следующей конструкции:

```
>>> - (5 ** 2)
-25
```

Возможно, и это еще не то, чего бы вам хотелось. Скобки помогут сделать конструкцию понятнее:

```
>>> (-5) ** 2
25
```

Таким образом, любому человеку, читающему код, не придется угадывать его предназначение и вспоминать правила приоритета.

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, 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». В отличие от римских цифр в арабских цифрах нет одного символа, который представлял 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 используете до того, как перенесете единицу. В двоичной (бинарной) системе счисления единственными цифрами являются 0 и 1. Такие значения может иметь и знаменитый *бит*. 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, помимо десятичной, вы можете выразить числа еще в трех системах счисления, используя следующие префиксы для целых чисел:

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

Основаниями здесь являются степени двойки. В некоторых случаях использование этих систем счисления может быть полезным, однако вполне вероятно, что вам никогда не понадобится ни одна другая система счисления, кроме старой доброй десятичной.

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 двоичное (основание 2) число 0b10, что означает *«одна (десятичная) двойка и ноль единиц»*:

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

Восьмеричное (основание 8) число 0o10 означает *«одна (десятичная) восьмерка и ноль единиц»*:

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

Шестнадцатеричное (основание 16) число 0x10 означает *«одно (десятичное) 16 и ноль единиц»*:

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

Вы можете пойти и в обратном направлении, преобразовав целое число в строку с любой из этих систем счисления:

```
>>> value = 65
>>> bin(value)
'0b1000001'
>>> oct(value)
'0o101'
>>> hex(value)
'0x41'
```

Функция `chr()` преобразует целое число в его строковый эквивалент, состоящий из одного символа:

```
>>> chr(65)
'A'
```

Функция `ord()` выполняет противоположную задачу:

```
>>> ord('A')
65
```

Если вам интересно, какие «цифры» использует шестнадцатеричная система счисления, то вот они: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e и f. 0xa — это десятичное число 10, а 0xf — десятичное 15. Добавьте 1 к 0xf и получите 0x10 (десятичное 16).

Зачем использовать системы счисления, отличные от десятичной? Они удобны для *битовых* операций, которые описаны в главе 12. Там же можно найти и детальную информацию о переводе чисел из одной системы счисления в другую.

У котов, как правило, есть по пять пальцев на передних лапах и по четыре на задних — всего 18. Если вы когда-нибудь встретите котов-ученых в лабораторных халатах, они, скорее всего, будут обсуждать арифметику восемнадцатеричной системы счисления. Мой кот Честер, которого вы можете увидеть на рис. 3.1, *полидакт* — у него целых 22 пальца (или около того — их сложно различить). Если бы он захотел подсчитать все кусочки корма, разбросанные вокруг его миски, он, вполне вероятно, воспользовался бы 22-ричной системой счисления (далее по тексту — *честеричной* системой счисления), в которой использовались бы цифры от 0 до 9 и символы от a до l.

Рис. 3.1. Честер — прекрасный пушистый парень, изобретатель честеричной системы счисления

Преобразования типов

Чтобы изменить тип данных на целочисленный, воспользуйтесь функцией `int()`.

Функция `int()` принимает один входной аргумент и возвращает одно значение — преобразованный в целое число эквивалент входного аргумента. При этом она сохраняет целую часть числа и отбрасывает любой остаток.

Как было сказано в начале главы, простейший тип данных в Python — *булев*, переменные этого типа могут иметь только значения `True` и `False`. После преобразования в целые числа они представляют собой значения `1` и `0`:

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

В свою очередь, функция `bool()` возвращает булев эквивалент целого числа:

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

При преобразовании числа с плавающей точкой в целое число просто отсекается все, что находится после десятичной точки:

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

Преобразование числа с плавающей точкой в булево значение не должно вас удивить:

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

Наконец, рассмотрим пример получения целочисленного значения из текстовой строки (глава 5), которая содержит только цифры и, возможно, разделители _ или знаки + и -:

```
>>> int('99')
99
>>> int('-23')
-23
>>> int('+12')
12
>>> int('1_000_000')
1000000
```

Если число в строке представлено не в десятичной системе счисления, а в какой-то иной, вы можете включить основание этой системы:

```
>>> int('10', 2) # двоичная
2
>>> int('10', 8) # восьмеричная
8
>>> int('10', 16) # шестнадцатеричная
16
>>> int('10', 22) # честеричная
22
```

Преобразование целого числа в целое число ничего не изменит, но и не навредит:

```
>>> 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 этом примере начинается с корректных символов-цифр (`99`), но продолжается символами, которые функция `int()` обработать не может.

Функция `int()` будет преобразовывать в целые числа строки, состоящие из цифр и чисел с плавающей точкой, но 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 попытается автоматически их преобразовать:

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

Булево значение `False` рассматривается как `0` или `0.0`, когда 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` имел ограничение в 32 или в 64 бита, в зависимости от вашего процессора. В 32 битах можно было хранить любое целое число в диапазоне от –2 147 483 648 до 2 147 483 647.

Для переменных типа `long` выделялось 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. Есть возможность поиграть и с такими большими числами, как *гугол* (это единица со ста последующими нулями), имя которому придумал девятилетний мальчик в 1920 году (https://ru.wikipedia.org/wiki/Гугол):

```
>>>
>>>googol = 10**100
>>> googol
10000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000
```

```
>>> googol * googol
10000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000
```

Гуголплекс — число, равное `10**googol` (если вы хотите записать его самостоятельно, то подскажу, что оно содержит тысячу нулей). Сначала именно так хотела назвать себя компания Google (https://www.quora.com/How-did-Google-get-its-name), но все же остановилась на варианте *«гугол»*. При этом правописание перед регистрацией доменного имени `google.com` никто не проверил.

Во многих других языках программирования такое число вызовет *переполнение целочисленного значения* и различные нежелательные последствия, так как компьютер не сможет предоставить все необходимое место. Python обрабатывает огромные целые числа порядка гугола без всяких проблем.

Числа с плавающей точкой

Значениями целочисленного типа являются целые числа, но числа *с плавающей точкой* (в Python они называются `float`) имеют десятичную точку:

```
>>> 5.
5.0
>>> 5.0
5.0
>>> 05.0
5.0
```

Числа с плавающей точкой могут содержать десятичную целочисленную экспоненту после буквы `e`:

```
>>> 5e0
5.0
>>> 5e1
50.0
>>> 5.0e1
50.0
>>> 5.0 * (10 ** 1)
50.0
```

Для разделения цифр можно использовать символ «нижнее подчеркивание» (_), как и в случае с целыми числами:

```
>>> million = 1_000_000.0
>>> million
1000000.0
>>> 1.0_0_1
1.001
```

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()`. Булевы значения по-прежнему обрабатываются как небольшие числа:

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

При преобразовании целого числа типа `int` в число типа `float` просто добавляется десятичная точка:

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

Можно конвертировать и строку с символами, представляющими собой корректные числа с плавающей точкой, — цифры, знаки, десятичную точку и знак `e` с показателем степени:

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

Когда вы работаете с целыми числами и числами с плавающей точкой, Python автоматически *преобразует* целочисленные значения в значения с плавающей точкой:

```
>>> 43 + 2.
45.0
```

Python также может преобразовывать булевы значения в целые числа и числа с плавающей точкой:

```
>>> False + 0
0
>>> False + 0.
0.0
>>> True + 0
1
>>> True + 0.
1.0
```

Математические функции

Python поддерживает комплексные числа и имеет привычный набор математических функций, таких как квадратный корень, косинус и т. д. Их рассмотрение отложим до главы 22, в которой также обсудим применение Python в науке.

Читайте далее

В следующей главе мы наконец попрощаемся с однострочными примерами кода. Вы научитесь принимать решения в коде с помощью утверждения `if`.

Упражнения

В этой главе мы рассмотрели атомы Python: числа, строки и переменные. Выполним с ними несколько небольших упражнений в интерактивном интерпретаторе.

3.1. Сколько секунд в часе? Используйте интерактивный интерпретатор как калькулятор и умножьте количество секунд в минуте (60) на количество минут в часе (тоже 60).

3.2. Присвойте результат вычисления предыдущего задания (секунды в часе) переменной, которая называется `seconds_per_hour`.

3.3. Сколько секунд в сутках? Используйте переменную `seconds_per_hour`.

3.4. Снова посчитайте количество секунд в сутках, но на этот раз сохраните результат в переменной `seconds_per_day`.

3.5. Разделите значение переменной `seconds_per_day` на значение переменной `seconds_per_hour`, используя деление с плавающей точкой (`/`).

3.6. Разделите значение переменной `seconds_per_day` на значение переменной `seconds_per_hour`, используя целочисленное деление (`//`). Совпадает ли полученный результат с ответом из предыдущего пункта без учета символов `.0` в конце?

ГЛАВА 4
Выбираем с помощью оператора if

О, если ты спокоен, не растерян,
Когда теряют головы вокруг...

Редьярд Киплинг. Если

В предыдущих главах вам встречались самые разные данные, но с ними вы практически не работали. В большинстве примеров использовался интерактивный интерпретатор, а сами они были достаточно короткими. Теперь вы увидите, как структурировать *код* Python, а не только данные.

В большинстве языков программирования такие символы, как фигурные скобки (`{` и `}`) и ключевые слова (например, `begin` и `end`), применяются для разбивки кода на разделы. В этих языках хорошим тоном является использование отбивки пробелами с целью сделать программу более легкочитаемой для себя и других. Существуют даже специальные инструменты, которые помогают красиво выстраивать код.

Гвидо ван Россум, разрабатывая Python, решил, что выделения пробелами достаточно для определения структуры программы, и отказался от ввода разнообразных 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, а тем, кто уже работал с другими языками программирования, такой подход может даже показаться странным. Однако с течением времени все становится на свои места и кажется абсолютно естественным. Вам даже понравится, что вы делаете больше, набирая меньше текста.

Наши первые примеры кода состояли из одной строки. А теперь посмотрим, как создавать комментарии и команды длиной несколько строк.

Комментируем с помощью символа

Комментарий — это фрагмент текста в вашей программе, который будет проигнорирован интерпретатором Python. Вы можете использовать комментарии, чтобы давать пояснение кода, делать какие-то пометки для себя или для чего-либо еще. Комментарий помечается символом `#`: все, что после `#` и 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 с/мин * 60 мин/ч * 24 ч/день
```

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

Python не дает возможности написать многострочный комментарий. Каждую строку или раздел комментария следует начинать 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 сравнительно короткие. Рекомендуемая (но не обязательная) максимальная длина строки равна 80 символам. Если вы не можете выразить свою мысль в рамках 80 символов, воспользуйтесь *символом продолжения* \ — просто поместите его в конце строки, и дальше Python будет действовать так, как будто это все та же строка.

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

```
>>> sum = 0
>>> sum += 1
>>> sum += 2
>>> sum += 3
>>> sum += 4
>>> sum
10
```

Но мог бы сделать и в одно действие, используя символ продолжения:

```
>>> sum = 1 + \
...       2 + \
...       3 + \
...       4
>>> sum
10
```

Если мы оставим обратный слеш в середине выражения, будет сгенерировано исключение:

```
>>> sum = 1 +
  File "<stdin>", line 1
    sum = 1 +
            ^
SyntaxError: invalid syntax
```

Есть небольшая хитрость: если использовать парные круглые (а также квадратные или фигурные) скобки, Python не будет жаловаться на некорректное окончание строк:

```
>>> sum = (
...     1 +
...     2 +
...     3 +
...     4)
>>>
>>> sum
10
```

В главе 5 вы также увидите, как парные тройные кавычки позволяют создавать многострочные выражения.

Сравниваем с помощью операторов if, elif и else

И вот мы готовы сделать первый шаг к *структурам кода*, которые вводят данные в программы. 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 выступают *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 экран.

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

Каждая строка `print()` имеет отступ под соответствующей проверкой. Я использовал отступ в четыре пробела для того, чтобы выделять подразделы. Вы можете использовать любое количество пробелов, однако 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 считать отступы.

Все выполненные в этом примере действия позже я объясню более детально.

- Булево значение `True` присваивается переменной `disaster`.
- Производится *условное сравнение* при помощи операторов `if` и `else` с выполнением разных фрагментов кода в зависимости от значений переменной `disaster`.
- *Вызывается функция* `print()` для вывода текста на экран.

Вы можете организовать проверку в проверке столько раз, сколько посчитаете нужным:

```
>>> furry = True
>>> large = True
>>> if furry:
...     if large:
...         print("It's a yeti.")
...     else:
...         print("It's a cat!")
... else:
...     if large:
...         print("It's a whale!")
...     else:
...         print("It's a human. Or a hairless cat.")
...
It's a yeti.
```

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 large`. Поскольку мы указали значение переменной `large` равным `True`, проверка вернет результат `True`, а следующая строка `else` будет проигнорирована. Это заставит Python запустить строку, размещенную под конструкцией `if large`, и вывести на экран строку `It's a yeti`.

Если нужно проверить больше двух вариантов, используйте оператор `if` для первого варианта, `elif` (это значит `else if` — «иначе если») для промежуточных и `else` для последнего:

```
>>> color = "mauve"
>>> 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 mauve
```

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 (`>=`).

Эти операторы возвращают булевы значения `True` или `False`. Посмотрим 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 нужно выполнить несколько сравнений одновременно, можно использовать *логические* (или *булевы*) *операторы* `and`, `or` и `not` для определения итогового логического результата.

Булевы операторы имеют более низкий *приоритет* по сравнению с фрагментами кода, которые они сравнивают. Это значит, что результаты фрагментов сначала вычисляются, а затем сравниваются. Поскольку в данном примере мы устанавливаем значение `x` равным `7`, проверка `5 < x` возвращает значение `True`, 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» на с. 73, 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 должно быть явно логическим `False`. Например, к `False` приравниваются 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 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!
```

При проверке выражения, а не простой переменной Python оценивает его значение и возвращает булев результат. Поэтому, если вы введете следующее:

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

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

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

Выполняем несколько сравнений с помощью оператора in

Предположим, что у вас есть буква и вы хотите узнать, является ли она гласной. Одним из вариантов решения может быть создание длинного утверждения `if`:

```
>>> letter = 'o'
>>> if letter == 'a' or letter == 'e' or letter == 'i' \
...     or letter == 'o' or letter == 'u':
...     print(letter, 'is a vowel')
... else:
...     print(letter, 'is not a vowel')
...
o is a vowel
>>>
```

Каждый раз, когда вам нужно выполнить множество подобных сравнений, разделенных ключевым словом `or`, используйте *оператор членства* `in`. В следующем примере мы проверяем, является ли буква гласной, более «питонским» способом, используя строку, состоящую целиком из гласных:

```
>>> vowels = 'aeiou'
>>> letter = 'o'
>>> letter in vowels
True
>>> if letter in vowels:
...     print(letter, 'is a vowel')
...
o is a vowel
```

Рассмотрим также небольшой пример того, как можно использовать оператор `in` для некоторых типов данных, о которых более подробно мы поговорим в нескольких следующих главах:

```
>>> letter = 'o'
>>> vowel_set = {'a', 'e', 'i', 'o', 'u'}
>>> letter in vowel_set
True
>>> vowel_list = ['a', 'e', 'i', 'o', 'u']
```

```
>>> letter in vowel_list
True
>>> vowel_tuple = ('a', 'e', 'i', 'o', 'u')
>>> letter in vowel_tuple
True
>>> vowel_dict = {'a': 'apple', 'e': 'elephant',
...                'i': 'impala', 'o': 'ocelot', 'u': 'unicorn'}
>>> letter in vowel_dict
True
>>> vowel_string = "aeiou"
>>> letter in vowel_string
True
```

В случае со словарем оператор `in` проверяет ключи (они находятся слева от двоеточия), а не значения.

Новое: I Am the Walrus[1]

В Python 3.8 появился *оператор-морж*, который выглядит следующим образом:

```
имя := выражение
```

Заметили моржа? (Он похож на смайлик с бивнями.)

Как правило, для присваивания и проверки нужно выполнить два шага:

```
>>> tweet_limit = 280
>>> tweet_string = "Blah" * 50
>>> diff = tweet_limit - len(tweet_string)
>>> if diff >= 0:
...     print("A fitting tweet")
... else:
...     print("Went over by", abs(diff))
...
A fitting tweet
```

С помощью новой силы клыков (операторов присваивания) мы можем два шага объединить в один:

```
>>> tweet_limit = 280
>>> tweet_string = "Blah" * 50
>>> if diff := tweet_limit - len(tweet_string) >= 0;
...     print("A fitting tweet")
... else:
...     print("Went over by", abs(diff))
...
A fitting tweet
```

Оператор-морж также может использоваться в циклах `for` и `while`, которые мы рассмотрим в главе 6.

---

[1] Название песни группы The Beatles. — *Примеч. пер.*

Читайте далее

Поговорим о строках и рассмотрим знаковые символы.

Упражнения

4.1. Выберите число от 1 до 10 и присвойте его переменной `secret`. Выберите еще одно число от 1 до 10 и присвойте его переменной `guess`. Далее напишите условные проверки (`if`, `else` и `elif`), чтобы вывести строку `'too low'`, если значение переменной `guess` меньше `7`, `'too high'`, если оно больше `7`, и `'just right'`, если равно `secret`.

4.2. Присвойте значения `True` или `False` переменным `small` и `green`. Напишите несколько утверждений `if/else`, которые выведут на экран информацию о том, соответствуют ли заданным условиям следующие растения: вишня, горошек, арбуз, тыква.

ГЛАВА 5
Текстовые строки

Мне всегда нравились странные персонажи.

Тим Бёртон

Благодаря книгам о компьютерах может сложиться впечатление, что программирование — это чистая математика. На самом деле большинство программистов гораздо чаще работают с текстовыми *строками*, а не с числами. В таком случае логическое (и креативное!) мышление бывает гораздо важнее математических навыков.

Строки являются первым примером *последовательностей* в Python. В частности, они представляют собой последовательности *символов*. Но что такое символ? Это наименьшая единица письменной системы, которая может быть буквой, цифрой, специальным символом, знаком препинания и даже пробелом или директивой вроде linefeeds. Символ определяется своим значением (тем, как он используется), а не тем, как он выглядит. У него может быть более одного визуального представления (в разных *шрифтах*), и, в свою очередь, одинаковое представление могут иметь несколько символов (например, символ `H` в латинском алфавите обозначает звук «х», а в кириллическом — «н»).

В этой главе рассматривается создание и форматирование простых текстовых строк на примере базового набора символов ASCII. Две важные темы, касающиеся текстов, обсуждаются в главе 12: символы *Unicode* (к этой теме относится ситуация с символом `H`, о которой я только что упомянул) и *регулярные выражения* (сравнение с шаблоном).

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 добиться подобного эффекта. Скоро вы узнаете, как это делается.

Создаем строки с помощью кавычек

Строка в Python создается заключением символов в парные одинарные или двойные кавычки:

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

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

В Python существует несколько особых типов строк, определяемых буквой, стоящей перед первой кавычкой. С буквы f или F начинается f-строка, которая используется для форматирования: она описана в конце главы. С буквы r или R начинается необработанная строка — она используется для предотвращения создания управляющих символов в строке (см. раздел «Создаем escape-последовательности с помощью символа \» на с. 94 и главу 12, в которой рассказывается об их использовании при проверке на соответствие шаблону). Возможна и комбинация fr (а также FR, Fr или fR), с которой начинается необработанная f-строка. Буквой u запускается строка из символов Unicode, аналогичная простой строке. С буквы b начинается значение объекта типа bytes (см. главу 12). Если я не упоминаю один из этих особых типов, значит, речь идет о старой доброй текстовой строке, содержащей символы Unicode.

Зачем нужны два вида кавычек? Основная идея заключается в том, чтобы позволить программисту создавать строки, содержащие символы кавычек. Вы можете пользоваться одинарными кавычками внутри строк, созданных с помощью двойных кавычек, и наоборот:

```
>>> "'Nay,' said the naysayer. 'Neigh?' said the horse."
"'Nay,' said the naysayer. 'Neigh?' said the horse."
>>> '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."
```

Допускается также использование трех одинарных (''') или трех двойных кавычек ("""):

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

Тройные кавычки не очень подходят для таких коротких строк. Обычно они используются для того, чтобы создавать *многострочный текст* наподобие следующего классического стихотворения Эдварда Лира:

```
>>> 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 пригла-

шения ... до тех пор, пока мы не ввели последние тройные кавычки и не перешли к следующей строке.)

Если вы попытаетесь создать стихотворение без тройных кавычек, Python начнет волноваться, когда вы перейдете 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 в строке. Если есть начальные или конечные 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()` выводит на экран содержимое строк и при этом удаляет из них кавычки. Она предназначена для удобства пользователя — услужливо добавляет пробел между каждым выводимым объектом, а также символ новой строки в конце:

```
>>> print ('Give', "us", '''some''', """space""")
Give us some space
```

Если вам не нужны пробелы или переход на новую строку, в главе 14 вы узнаете, как от них избавиться.

Интерактивный интерпретатор выводит строку с одинарными кавычками и *управляющими символами*, такими как `\n`: это объясняется в разделе «Создаем escape-последовательности с помощью символа \» далее в этой главе.

```
>>> """'Guten Morgen, mein Herr!'
... said mad king Ludwig to his wig."""
"'Guten Morgen, mein Herr!'\nsaid mad king Ludwig to his wig."
```

Наконец, вам может понадобиться поработать с *пустой строкой*. В ней нет символов, но она совершенно корректна. Пустая строка создается с помощью любых упомянутых ранее кавычек:

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

Создаем строки с помощью функции str()

Создать строку из другого типа данных можно с помощью функции `str()`:

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

В Python функция `str()` используется также для внутренних нужд: например, при вызове функции `print()` для объектов, которые не являются строками, и при *форматировании строк*, о чем мы поговорим далее в этой главе.

Создаем escape-последовательности с помощью символа \

Python позволяет использовать *escape-последовательности* внутри строк, чтобы добиться эффекта, который по-другому выразить было бы трудно. Символ с размещенным перед ним обратным слешем (`\`) получает особое значение. Наиболее распространена последовательность `\n`, которая означает переход на новую строку. С ее помощью можно создавать многострочные строки из однострочных:

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

Последовательность `\t` (табуляция) 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, но увидеть ее вы, конечно, не можете.)

Кроме того, вам могут понадобиться последовательности \' или \", чтобы поместить одинарные или двойные кавычки в строку, заключенную в такие же символы:

```
>>> testimony = "\"I did nothing!\" he said. \"Or that other thing.\""
>>> testimony
'"I did nothing!" he said. "Or that other thing."'
>>> print(testimony)
"I did nothing!" he said. "Or that other thing."

>>> fact = "The world's largest rubber duck was 54'2\" by 65'7\" by 105'"
>>> print(fact)
The world's largest rubber duck was 54'2" by 65'7" by 105'
```

А если вам нужен обратный слеш, просто напечатайте два (первый из них экранирует второй):

```
>>> speech = 'The backslash (\\) bends over backwards to please you.'
>>> print(speech)
The backslash (\) bends over backwards to please you.
>>>
```

Как я упоминал ранее в этой главе, в необработанной строке escape-символы не работают:

```
>>> info = r'Type a \n to get a new line in a normal string'
>>> info
'Type a \\n to get a new line in a normal string'
>>> print(info)
Type a \n to get a new line in a normal string
```

(Интерактивный интерпретатор добавил дополнительный обратный слеш при первом отображении значения переменной `info`.)

В необработанных строках работают все реальные (а не выполненные с помощью последовательности `'\n'`) переходы на новую строку:

```
>>> poem = r'''Boys and girls, come out to play.
... The moon doth shine as bright as day.'''
>>> poem
'Boys and girls, come out to play.\nThe moon doth shine as bright as day.'
>>> print(poem)
Boys and girls, come out to play.
The moon doth shine as bright as day.
```

Объединяем строки с использованием символа +

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

```
>>> 'Release the kraken! ' + 'No, wait!'
'Release the kraken! 'No, wait!'
```

Строки (не переменные) можно объединять, просто расположив их одну за другой:

```
>>> "My word! " "A gentleman caller!"
'My word! A gentleman caller!'
>>> "Alas! ""The kraken!"
'Alas! The kraken!'
```

Если подобных действий много, избежать создания переходов на новую строку можно, заключив строки в скобки:

```
>>> vowels = ( 'a'
... "e" '''i'''
... 'o' """u"""
... )
>>> vowels
'aeiou'
```

Python сам *не* добавляет пробелы при конкатенации строк, поэтому в более ранних примерах нам нужно было явно проставлять пробелы. Python добавляет пробел между каждым аргументом выражения `print()` и символом новой строки в конце:

```
>>> a = 'Duck.'
>>> b = a
>>> c = 'Grey Duck!'
>>> a + b + c
'Duck. Duck. Grey Duck!'
>>> print(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 посмотрите, что получится:

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

Обратите внимание на то, что оператор * имеет более высокий приоритет, чем +, поэтому строка будет размножена до того, как к ней добавится символ перехода на новую строку.

Извлекаем символ с помощью символов [ ]

Для получения одного символа из строки укажите его *смещение* в квадратных скобках после имени строки. Смещение первого (крайнего слева) символа равно 0, следующего — 1 и т. д. Чтобы не считать, смещение последнего (крайнего справа) символа можно указать как –1. 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 идут от 0 к длине –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 я рассмотрю в главе 7.

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 (совсем скоро вы ее увидите), таких как `replace()` или `slice`:

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

Мы не изменили значение переменной `name`. Интерактивный интерпретатор просто вывел на экран результат замены.

Извлекаем подстроки, используя разделение

Из строки можно извлечь *подстроку* (часть строки) с помощью *разделения*.

Вы задаете разделение с помощью квадратных скобок, смещения начала подстроки *начало* и смещения конца подстроки *конец*, а также опционального размера шага между ними *шаг*. Некоторые из этих параметров могут быть пропущены. В подстроку будут включены символы, расположенные от точки, на которую указывает смещение *начало*, до точки, на которую указывает смещение *конец*.

- Оператор `[:]` извлекает всю последовательность от начала до конца.
- Оператор `[`*начало*`:]` извлекает последовательность с точки, на которую указывает смещение *начало*, до конца.
- Оператор `[:`*конец*`]` извлекает последовательность от начала до точки, на которую указывает смещение *конец* минус 1.
- Оператор `[`*начало*`:`*конец*`]` извлекает последовательность с точки, на которую указывает смещение *начало*, до точки, на которую указывает смещение *конец* минус 1.
- Оператор `[`*начало*`:`*конец*`:`*шаг*`]` извлекает последовательность с точки, на которую указывает смещение *начало*, до точки, на которую указывает смещение *конец* минус 1, опуская символы, чье смещение внутри подстроки кратно *шаг*.

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 *начало*, функция будет использовать в качестве его значения `0` (начало строки). Если не укажете смещение *конец*, его значением будет считаться 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'
```

Так — начиная с 10-го и заканчивая последним:

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

А теперь получим символы с 12-го по 14-й. Python не включает в разделение конечное смещение. Стартовое смещение *включено*, а конечное — *не включено*:

```
>>> 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, как этот пример отличается от предыдущего, в котором старт с позиции –3 приводил к букве x. В этом примере конец диапазона –3 означает, что последней будет буква со смещением –4 — w:

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

В этом 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 *конец* должно быть на единицу больше, чем реальное смещение.)

И это еще не все! Если задать отрицательный шаг, удобный разделитель Python дает возможность 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, тот же результат можно получить и другим способом:

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

Операция *разделения* более лояльно относится к некорректно указанным смещениям, чем поиск по индексу с помощью символов `[]`. 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]
''
```

Измеряем длину строки с помощью функции len()

До этого момента мы использовали специальные знаки, такие как `+`, для работы со строками. Но их не так уж и много. Теперь начнем использовать некоторые из встроенных *функций* Python: именованные фрагменты кода, которые выполняют определенные операции.

Функция `len()` подсчитывает количество символов в строке:

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

Функцию `len()` можно использовать и для других типов последовательностей, о чем мы поговорим в главе 7.

Разделяем строку с помощью функции split()

В отличие от `len()`, некоторые функции применимы только к строкам. Для того чтобы использовать строковую функцию, следует ввести имя строки, точку, имя функции и *аргументы*, которые нужны функции: *`строка.функция(аргументы)`*. Более подробно об этом поговорим в главе 9.

Встроенная функция `split()` позволяет разбить строку на *список* строк меньшего размера с использованием *разделителя*. Со списками вы познакомитесь в следующей главе. Список — это последовательность значений, разделенных запятыми и заключенных в квадратные скобки:

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

В предыдущем примере строка имела имя `tasks`, а строковая функция называлась `split()` и имела один аргумент-разделитель `','`. Если вы не укажете разделитель,

функция `split()` будет использовать любую последовательность пробелов, а также символы новой строки и табуляцию:

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

При вызове функции `split` без аргументов круглые скобки все равно надо ставить — именно так Python понимает, что вы вызываете функцию.

Объединяем строки с помощью функции join()

Легко догадаться, 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)`. Таким образом, чтобы объединить строки списка `lines`, разделенные символами новой строки, вам нужно написать `'\n'.join(lines)`. В следующем примере мы объединим в список несколько имен, разделив их 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
```

Заменяем символы с использованием функции replace()

Функция `replace()` предназначена для замены одной подстроки другой. Вы указываете старую подстроку, новую подстроку и то, сколько экземпляров старой подстроки нужно заменить. Если вы пропустите последний аргумент, функция заменит все экземпляры. В этом примере только одна строка (`'duck'`) сопоставляется и заменяется:

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

Здесь заменим до 100 экземпляров:

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

Если вы точно знаете, какую подстроку или подстроки хотите заменить, функция `replace()` окажется для вас хорошим выбором. Но будьте осторожны: если бы во втором примере вы заменили строку из одного символа `'a'`, а не строку из двух символов `'a '` (после `a` идет пробел), символы `'a'` заменились бы и в середине слов:

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

Иногда вам нужно убедиться в том, что подстрока является целым словом, началом слова и т. п. В такой ситуации понадобятся *регулярные выражения*. Они подробно описаны в главе 12.

Устраняем символы с помощью функции strip()

Нередко требуется убрать из строки начальные или конечные символы отступа, чаще всего пробелы. Представленная здесь функция `strip()` подразумевает, что вы хотите избавиться от символов-пробелов (`' '`, `'\t'`, `'\n'`), если только вы не предоставите ей аргумент. Функция `strip()` удаляет символы с обоих концов строки, `lstrip()` — только в начале, а `rstrip()` — только в конце. Давайте представим, что строковая переменная `world` содержит строку `earth`, окруженную пробелами:

```
>>> world = "    earth   "
>>> world.strip()
'earth'
>>> world.strip(' ')
'earth'
>>> world.lstrip()
'earth   '
>>> world.rstrip()
'    earth'
```

Если искомого символа в строке нет, ничего не происходит:

```
>>> world.strip('!')
'    earth   '
```

Кроме отсутствия аргумента (когда подразумевается, что нужно найти пробелы) или отдельного символа, возможна и ситуация, когда функция `strip()` удалит все символы из последовательности:

```
>>> blurt = "What the...!!?"
>>> blurt.strip('.?!')
'What the'
```

В приложении Д представлены отдельные определения групп символов, которые можно использовать для передачи в функцию `strip()`:

```
>>> import string
>>> string.whitespace
' \t\n\r\x0b\x0c'
>>> string.punctuation
'!"#$%&\'()*+,-./:;<=>?@[\\]^_`{|}~'
>>> blurt = "What the...!!?"
>>> blurt.strip(string.punctuation)
'What the'
>>> prospector = "What in tarnation ...??!!"
>>> prospector.strip(string.whitespace + string.punctuation)
'What in tarnation'
```

Поиск и выбор

Python содержит большой набор функций для работы со строками. Рассмотрим принцип работы самых распространенных из них. Тестовым объектом станет следующая строка, содержащая текст бессмертного стихотворения «What Is Liquid?» Маргарет Кавендиш, герцогини 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.'''
```

Вдохновляет!

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 All?

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

Заканчивается ли оно фразой That's all, folks!?

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

В Python есть два метода поиска смещения подстроки — `find()` и `index()`. У каждого метода имеется возможность начинать поиск или с начала, или с конца. В случае если подстрока присутствует в строке, они работают одинаково. Если подстрока не найдена, метод `find()` возвращает `-1`, а метод `index()` генерирует исключение.

Найдем смещение первого появления в стихотворении слова `the`:

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

А теперь — последнего:

```
>>> word = 'the'
>>> poem.rfind(word)
214
>>> poem.rindex(word)
214
```

Но что, если подстроки нет?

```
>>> word = "duck"
>>> poem.find(word)
-1
>>> poem.rfind(word)
-1
>>> poem.index(word)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: substring not found
>>> poem.rfind(word)
-1
>>> poem.rindex(word)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: substring not found
```

Сколько раз встречается трехбуквенное сочетание `the`?

```
>>> word = 'the'
>>> poem.count(word)
3
```

Являются ли все символы в стихотворении буквами или цифрами?

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

Нет, в стихотворении имеются еще и знаки препинания.

Регистр

В этом разделе мы рассмотрим еще несколько примеров использования встроенных функций. В качестве тестовой 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 в одном из этих примеров строка setup на самом деле не изменяется. Просто берется значение переменной setup, над ним выполняется некоторое действие, а затем результат возвращается в виде новой строки.

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 строчными буквами:

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

Сменим регистры букв на противоположные:

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

Выравнивание

Теперь поработаем с функциями выравнивания. Строка выравнивается в пределах указанного общего количества пробелов (в данном примере 30).

Отцентрируем 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...'
```

Далее рассмотрим другие способы выравнивания строки.

Форматирование

Вы уже видели, как можно *конкатенировать* строки с помощью оператора `+`. А теперь узнаете, как *интерполировать* данные в строки, используя разные форматы. Этим приемом можно пользоваться, чтобы создавать отчеты и другие документы, для которых важен определенный внешний вид.

Помимо функций, рассмотренных в предыдущем разделе, Python имеет три способа форматирования строк:

- *старый стиль* (поддерживается в Python 2 и 3);
- *новый стиль* (Python 2.6 и выше);
- *f-строки* (Python 3.6 и выше).

Старый стиль: %

Старый стиль форматирования строк имеет форму *строка % данные*. Внутри строки находятся интерполяционные последовательности. В табл. 5.1 показано, что самая простая последовательность — это символ `%` и буква. Буква указывает на тип данных, которые должны быть отформатированы.

Таблица 5.1. 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 %

Вы можете использовать последовательность `%s` для любого типа данных — Python отформатирует их как строку без дополнительных пробелов.

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 нужно интерполировать строку. Количество использованных символов `%` должно совпадать с количеством объектов, расположенных после `%`. Один элемент, такой как `actor`, располагается сразу после символа `%`. Если таких объектов несколько, они должны быть сгруппированы в *кортеж* (читайте подробнее в главе 7), то есть их нужно заключить в скобки, разделив запятыми. Например, `(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, а также выравнивание и заполнение символами. Эти конструкции похожи на своего рода небольшой язык, достаточно ограниченный по сравнению с тем, который будет представлен в следующих двух подразделах. Какие это значения:

- в начале располагается символ `'%'`;
- опциональный символ *выравнивания*: ничего или `'+'` означают выравнивание по правому краю, а `'-'` — по левому;
- опциональное поле *мин_ширина*, в котором указывается длина строки;
- опциональный символ `'.'`, разделяющий поля *мин_ширина* и *макс_символы*;
- опциональное поле *макс_символы* (используется в том случае, если тип преобразования `s`), в котором указывается, сколько символов значения нужно вывести на экран. Если тип преобразования `f`, в этом поле указывается *точность* (сколько символов выводится после десятичной точки);
- символ, определяющий *тип преобразования*, из табл. 5.1.

Выглядит запутанно. Рассмотрим примеры для строки:

```
>>> thing = 'woodchuck'
>>> '%s' % thing
'woodchuck'
>>> '%12s' % thing
'   woodchuck'
>>> '%+12s' % thing
'   woodchuck'
>>> '%-12s' % thing
'woodchuck   '
>>> '%.3s' % thing
'woo'
>>> '%12.3s' % thing
'         woo'
>>> '%-12.3s' % thing
'woo         '
```

Теперь рассмотрим примеры для чисел с плавающей точкой (`%f`):

```
>>> thing = 98.6
>>> '%f' % thing
'98.600000'
>>> '%12f' % thing
'   98.600000'
>>> '%+12f' % thing
'  +98.600000'
>>> '%-12f' % thing
'98.600000   '
>>> '%.3f' % thing
'98.600'
>>> '%12.3f' % thing
'      98.600'
>>> '%-12.3f' % thing
'98.600      '
```

А теперь — для целых чисел (`%d`):

```
>>> thing = 9876
>>> '%d' % thing
'9876'
>>> '%12d' % thing
'        9876'
>>> '%+12d' % thing
'       +9876'
>>> '%-12d' % thing
'9876        '
>>> '%.3d' % thing
'9876'
>>> '%12.3d' % thing
'        9876'
>>> '%-12.3d' % thing
'9876        '
```

Для целых чисел конструкция `%+12d` заставляет вывести знак числа, при этом строки формата, содержащие конструкцию `.3`, ни на что не влияют, поскольку предназначены для чисел с плавающей точкой.

Новый стиль: используем символы {} и функцию format()

Старый стиль форматирования все еще поддерживается, и в Python 2, который остановился на версии 2.7, он будет поддерживаться всегда. В Python 3 применяется новый стиль форматирования. Если у вас установлен Python 3.6 или выше, рекомендую воспользоваться *f-строками* (см. подраздел «Самый новый стиль: f-строки» далее в этой главе).

Новый стиль форматирования имеет вид *`строка`*`.format(`*`данные`*`)`.

Строка формата выглядит немного иначе, чем в предыдущем подразделе. Простейший пример использования этого стиля показан здесь:

```
>>> thing = 'woodchuck'
>>> '{}'.format(thing)
'woodchuck'
```

Аргументы функции `format()` должны идти в том порядке, в котором расставлены заполнители `{}` в строке формата:

```
>>> thing = 'woodchuck'
>>> place = 'lake'
>>> 'The {} is in the {}.'.format(thing, plкace)
'The woodchuck is in the lake.'
```

В новом стиле форматирования вы также можете указать позицию аргументов следующим образом:

```
>>> 'The {1} is in the {0}.'.format(place, thing)
'The woodchuck is in the lake.'
```

Значение `0` относится к первому аргументу `place`, а `1` — к `thing`.

Аргументы функции `format()` могут быть именованными:

```
>>> 'The {thing} is in the {place}'.format(thing='duck', place='bathtub')
'The duck is in the bathtub'
```

Они также могут быть словарями:

```
>>> d = {'thing': 'duck', 'place': 'bathtub'}
```

В следующем примере `{0}` — это первый аргумент функции `format()` (словарь `d`):

```
>>> 'The {0[thing]} is in the {0[place]}.'.format(d)
'The duck is in the bathtub.'
```

В этих примерах аргументы выводились в формате, установленном по умолчанию. В новом стиле форматирования строка формата несколько отличается от той, которая использовалась в старом стиле. Ее отличия:

- начальное двоеточие (`':'`);
- опциональный символ-заполнитель (по умолчанию `' '`), которым заполняется строка, если ее длина меньше, чем *`мин_ширина`*;
- опциональный символ *выравнивания*. В этот раз вариантом по умолчанию является выравнивание по левому краю. Символ `'<'` означает выравнивание по левому краю, символ `'>'` — по правому, а символ `'^'` означает выравнивание по центру;
- опциональный *знак* для чисел. Отсутствие значения приведет к тому, что знак будет отображаться только для отрицательных чисел. Символ `' '` означает, что для отрицательных чисел будет добавляться знак `-`, а для положительных — пробел (`' '`);
- необязательное поле *`мин_ширина`*. Необязательный символ (`'.'`) используется для отделения значений полей *`мин_ширина`* и *`макс_символы`*;
- необязательное поле *`макс_символы`*;
- *тип преобразования*.

```
>>> thing = 'wraith'
>>> place = 'window'
>>> 'The {} is at the {}'.format(thing, place)
'The wraith is at the window'
>>> 'The {:10s} is at the {:10s}'.format(thing, place)
'The wraith     is at the window    '
```

```
>>> 'The {:<10s} is at the {:<10s}'.format(thing, place)
'The wraith     is at the window    '
>>> 'The {:^10s} is at the {:^10s}'.format(thing, place)
'The   wraith   is at the   window  '
>>> 'The {:>10s} is at the {:>10s}'.format(thing, place)
'The     wraith is at the     window'
>>> 'The {:!^10s} is at the {:!^10s}'.format(thing, place)
'The !!wraith!! is at the !!window!!'
```

Самый новый стиль: f-строки

На данный момент для форматирования строк рекомендуется использовать *f-строки*, которые появились в версии Python 3.6.

Чтобы создать f-строку, нужно сделать следующее:

- ввести букву `f` или `F` перед первой кавычкой;
- поместить имена переменной или выражения в фигурные скобки (`{}`), чтобы их значения попали в строку.

Этот способ форматирования похож на новый стиль, но здесь не задействованы функция `format()`, пустые скобки (`{}`) и позиционные аргументы (`{1}`) в строке формата.

```
>>> thing = 'wereduck'
>>> place = 'werepond'
>>> f'The {thing} is in the {place}'
'The wereduck is in the werepond'
```

Как я уже говорил, в фигурных скобках можно размещать и выражения:

```
>>> f'The {thing.capitalize()} is in the {place.rjust(20)}'
'The Wereduck is in the             werepond'
```

Таким образом, то, что вы могли делать внутри функции `format()` из предыдущего раздела, можно сделать и здесь внутри фигурных скобок в главной строке. Это упрощает чтение.

Для f-строк используется такой же язык форматирования (ширина, заполнитель, выравнивание), как и в новом стиле. Выражения размещаются после двоеточия:

```
>>> f'The {thing:>20} is in the {place:.^20}'
'The             wereduck is in the ......werepond......'
```

Начиная с версии Python 3.8, f-строки позволяют выводить не только значения переменных, но и их имена. Это очень удобно при отладке. Идея заключается в том, чтобы поставить знак `=` после имени переменной, размещенного в фигурных скобках:

```
>>> f'{thing =}, {place =}'
thing = 'wereduck', place = 'werepond'
```

Этот прием также работает и для выражений. Каждое выражение будет выведено полностью:

```
>>> f'{thing[-4:] =}, {place.title() =}'
thing[-4:] = 'duck', place.title() = 'Werepond'
```

Наконец, после знака = можно поставить двоеточие, за которым будут следовать аргументы, такие как длины и выравнивания:

```
>>> f'{thing = :>4.4}'
thing = 'were'
```

Что еще можно делать со строками

В Python имеется гораздо больше функций для работы со строками, нежели я сейчас описал. Некоторые из них мы рассмотрим в следующих главах (особенно в главе 12), но описания всех можно найти в стандартной документации (http://bit.ly/py-docs-strings).

Читайте далее

Следующая глава посвящена циклам в Python.

Упражнения

5.1. Напишите с заглавной буквы слово, которое начинается с буквы m:

```
>>> song = """When an eel grabs your arm,
... And it causes great harm,
... That's - a moray!"""
```

5.2. Выведите на экран все вопросы из списка, а также правильные ответы в таком виде:

Q: *вопрос*

A: *ответ*

```
>>> questions = [
...     "We don't serve strings around here. Are you a string?",
...     "What is said on Father's Day in the forest?",
...     "What makes the sound 'Sis! Boom! Bah!'?"
...     ]
>>> answers = [
...     "An exploding sheep.",
...     "No, I'm a frayed knot.",
...     "'Pop!' goes the weasel."
...     ]
```

5.3. Выведите на экран следующее стихотворение, используя старый стиль форматирования. Подставьте в него такие строки: `'roast beef'`, `'ham'`, `'head'` и `'clam'`:

```
My kitty cat likes %s,
My kitty cat likes %s,
My kitty cat fell on his %s
And now thinks he's a %s.
```

5.4. Напишите письмо с использованием нового стиля форматирования. Сохраните предложенную строку в переменной `letter` (она понадобится вам в упражнении ниже):

```
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}
```

5.5. Присвойте значения переменным `'salutation'`, `'name'`, `'product'`, `'verbed'` (глагол в прошедшем времени), `'room'`, `'animals'`, `'percent'`, `'spokesman'` и `'job_title'`. С помощью функции `letter.format()` выведите на экран значение переменной `letter`, в которую подставлены эти значения.

5.6. После проведения публичных опросов с целью выбора имени появились: английская подводная лодка Boaty McBoatface, австралийская беговая лошадь Horsey McHorseface и шведский поезд Trainy McTrainface. Используйте форматирование с символом `%` для того, чтобы вывести на экран победившие имена для утки, тыквы и шпица.

5.7. Сделайте то же самое с помощью функции `format()`.

5.8. А теперь еще раз с использованием *f-строк*.

ГЛАВА 6

Создаем циклы с помощью ключевых слов while и for

> При всем при том, при всем при том, пускай бедны мы с вами...
>
> *Роберт Бёрнс. Честная бедность*

Проверки с помощью `if`, `elif` и `else` выполняются последовательно, одна за другой. Иногда определенные операции требуется выполнить более чем один раз. Для этого нужен *цикл*, и Python предлагает два варианта: `while` и `for`.

Повторяем действия с помощью цикла while

Самым простым циклическим механизмом 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 началу цикла и снова сравнивает значение переменной `count` с числом `5`. Значение переменной `count` теперь равно `2`, весь цикл `while` проходит очередной раз, и затем переменная `count` увеличивается до `3`.

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

Прерываем цикл с помощью оператора break

Если вы хотите, чтобы цикл выполнялся до тех пор, пока что-то не произойдет, однако вы не знаете точно, когда это событие случится, можно воспользоваться *бесконечным циклом*, содержащим оператор `break`. В этот раз мы прочитаем строку с клавиатуры с помощью функции `input()`, а затем выведем ее на экран, сделав первую букву прописной. Цикл прервется, 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, используя оператор continue

Иногда вам нужно не прерывать весь цикл, а пропустить по какой-либо причине только одну итерацию. Рассмотрим пример: прочитаем целое число, и если оно четное, выведем на экран его значение в квадрате, если же нечетное — пропустим его. 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 раньше, с помощью блока else

Если цикл `while` завершился нормально (без вызова `break`), управление передается в опциональный блок `else`. Вы пользуетесь этим, когда кодируете цикл так, чтобы, выполняя некоторую проверку, он прервался, как только проверка успешно выполнится. Блок `else` запустится в том случае, если цикл `while` пройдет полностью, но искомый объект так и 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 не вызываем
...     print('No even number found')
...
No even number found
```

Такое использование ключевого слова else может показаться нелогичным. Воспринимайте его как проверку на прерывание цикла.

Выполняем итерации с использованием ключевых слов for и in

В Python *итераторы* часто используются по одной простой причине. Они позволяют вам проходить структуры данных без знания того, насколько эти структуры велики и как реализованы. Вы даже можете обратиться к данным, которые были созданы во время работы программы. Это позволяет обрабатывать *потоки* данных, которые в противном случае не поместились бы в память компьютера.

Чтобы продемонстрировать итерирование, нам нужно что-то, по чему можно проитерировать. Вы уже познакомились со строками в главе 5, но пока не знаете всей информации о других *итерабельных* объектах, таких как списки и кортежи (глава 7) или словари (глава 8). Сейчас я покажу два способа итерирования по строке, а итерирование по другим типам рассмотрим в соответствующих главах.

Вполне возможно пройти по строке таким образом:

```
>>> word = 'thud'
>>> offset = 0
>>> while offset < len(word):
...     print(word[offset])
...     offset += 1
...
t
h
u
d
```

Однако существует более характерный для Python способ решения этой задачи:

```
>>> for letter in word:
...     print(letter)
...
t
h
u
d
```

Итерирование по строке возвращает один символ за раз.

Прерываем цикл с помощью оператора break

Оператор `break` прерывает цикл `for` точно так же, как и цикл `while`.

```
>>> word = 'thud'
>>> for letter in word:
...     if letter == 'u':
...         break
...     print(letter)
...
t
h
```

Пропускаем итерации, используя оператор 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 раньше, с помощью блока 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 в тех случаях, когда вам нужно убедиться в том, что предыдущий цикл выполнился полностью, а не был прерван `break`.

```
>>> word = 'thud'
>>> for letter in word:
...     if letter == 'x':
...         print("Eek! An 'x'!")
...         break
...     print(letter)
... else:
...     print("No 'x' in there.")
...
t
h
u
d
No 'x' in there
```

Как и в цикле while, в цикле for использование блока else может показаться нелогичным. Можно рассматривать цикл for как поиск чего-либо, а появление else — как сигнал, что вы ничего не нашли. Чтобы получить тот же эффект без блока else, используйте переменную, которая будет показывать, нашелся ли искомый элемент в цикле for.

Генерируем числовые последовательности с помощью функции range()

Функция `range()` выбирает поток чисел в заданном диапазоне без необходимости создавать и сохранять крупную структуру данных, такую как список или кортеж. Это позволяет создавать большие диапазоны, не используя всю память на вашем компьютере и не сбивая программу.

Функцию `range()` можно применять так же, как функцию `slice()`: `range(`*`начало, конец, шаг`*`)`. Если вы опустите значение *`начало`*, диапазон начнется с `0`. Обязательным является только значение *`конец`*: как и в случае со `slice()`, последнее учтенное значение будет прямо перед *`конец`*. Значение по умолчанию *`шаг`* 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, такую как список. 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 шаг равен 2, что позволяет получить все четные числа от 0 до 10:

```
>>> list(range(0, 11, 2))
[0, 2, 4, 6, 8, 10]
```

Прочие итераторы

В главе 14 рассматривается итерирование по файлам. В главе 10 вы увидите, как использовать итерирование для объектов, которые вы сами определили. В главе 11 мы рассмотрим `itertools` — стандартный модуль Python, содержащий множество удобных функций.

Читайте далее

В следующей главе мы объединим индивидуальные объекты в *списки* и *кортежи*.

Упражнения

6.1. Используйте цикл `for`, чтобы вывести на экран значения списка `[3, 2, 1, 0]`.

6.2. Присвойте значение `7` переменной `guess_me` и значение `1` переменной `number`. Напишите цикл `while`, который сравнивает переменные `number` и `guess_me`. Выведите строку `'too low'`, если значение переменной `start` меньше значения переменной `guess_me`. Если значение переменной `number` равно значению переменной `guess_me`, выведите строку `'found it!'` и выйдите из цикла. Если значение переменной `number` больше значения переменной `guess_me`, выведите строку `'oops'` и выйдите из цикла. Увеличьте значение переменной `number` на выходе из цикла.

6.3. Присвойте значение `5` переменной `guess_me`. Используйте цикл `for` для того, чтобы проитерировать с помощью переменной `number` по диапазону `range(10)`. Если значение переменной `number` меньше, чем значение `guess_me`, выведите на экран сообщение `'too low'`. Если оно равно значению `guess_me` — выведите сообщение `'found it!'`, а затем выйдите из цикла. Если значение переменной `number` больше, чем `guess_me`, выведите на экран сообщение `'oops'` и выйдите из цикла.

ГЛАВА 7
Кортежи и списки

> Человек отличается от низших приматов страстью
> к составлению списков.
>
> *Гарри Аллен Смит*

В предыдущих главах мы говорили о базовых типах данных Python, таких как булевы значения, целочисленные значения, числа с плавающей точкой и строки. Если представлять их как атомы, то структуры данных, которые мы рассмотрим в этой главе, можно назвать молекулами. Так и есть: мы объединим базовые типы в более сложные структуры, которые вы будете использовать каждый день. Бóльшая часть работы программиста состоит из «разрезания» данных и «склеивания» их в конкретные формы, поэтому сейчас вы узнаете, как пользоваться ножовками и клеевыми пистолетами.

Большинство языков программирования могут представлять последовательность в виде объектов, проиндексированных по их позиции, выраженной целым числом: первый, второй и далее до последнего. Вы уже знакомы со *строками* — последовательностями символов.

В Python есть еще две структуры-последовательности: *кортежи* и *списки*. Они могут содержать ноль и более элементов. В отличие от строк в кортежах и списках допускаются элементы разных типов: по факту 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 включаете в кортеж элемент (всего один раз), он «запекается» и больше не изменяется. Списки же можно *изменять* — добавлять и удалять элементы в любой удобный момент. Я покажу вам множество примеров использования обоих типов, сделав акцент на списках.

Кортежи

Давайте сразу же рассмотрим один очевидный вопрос. Вы могли слышать два возможных варианта произношения слова *tuple* (кортеж). Какой же из них является правильным? Гвидо ван Россум, создатель языка Python, написал в Twitter

(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 об этом не говорю :)».

Создаем кортежи с помощью запятых и оператора ()

Синтаксис создания кортежей несколько необычен, что вы и увидите в следующих примерах.

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

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

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

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

Вы можете поместить элемент в круглые скобки и получить такой же кортеж:

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

Однако следует иметь в виду: если в круглые скобки вы поместите один объект и опустите при этом запятую, в результате вы получите не кортеж, а тот же самый объект (в этом примере строку `'Groucho'`):

```
>>> one_marx = ('Groucho')
>>> one_marx
'Groucho'
>>> type(one_marx)
<class 'str'>
```

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. Как правило, они не нужны для определения кортежа, но с ними более безопасно, так как они делают кортеж более заметным:

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

В тех случаях, когда у запятой могут быть и другие варианты использования, также рекомендуется ставить круглые скобки. В следующем примере вы можете создать кортеж с одним элементом и присвоить ему значение, поставив в конце запятую, но не можете передать эту конструкцию как аргумент функции:

```
>>> one_marx = 'Groucho',
>>> type(one_marx)
<class 'tuple'>
>>> type('Groucho',)
<class 'str'>
>>> type(('Groucho',))
<class 'tuple'>
```

Кортежи позволяют присваивать значение нескольким переменным одновременно:

```
>>> 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 для обмена значениями с помощью одного выражения, не применяя временную переменную:

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

Создаем кортежи с помощью функции tuple()

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

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

Объединяем кортежи с помощью оператора +

Это похоже на объединение строк:

```
>>> ('Groucho',) + ('Chico', 'Harpo')
('Groucho', 'Chico', 'Harpo')
```

Размножаем элементы с помощью оператора *

Принцип похож на многократное использование оператора +:

```
>>> ('yada',) * 3
('yada', 'yada', 'yada')
```

Сравниваем кортежи

Сравнение кортежей похоже на сравнение списков:

```
>>> a = (7, 2)
>>> b = (7, 2, 9)
>>> a == b
False
>>> a <= b
True
>>> a < b
True
```

Итерируем по кортежам с помощью for и in

Итерирование по кортежам выполняется так же, как и итерирование по другим типам:

```
>>> words = ('fresh','out', 'of', 'ideas')
>>> for word in words:
...     print(word)
...
fresh
out
of
ideas
```

Изменяем кортеж

Этого сделать вы не можете! Как и строки, кортежи неизменяемы. Но ранее вы уже видели на примере строк, что можно *сконкатенировать* (объединить) кортежи и создать таким образом новый кортеж:

```
>>> t1 = ('Fee', 'Fie', 'Foe')
>>> t2 = ('Flop',)
>>> t1 + t2
('Fee', 'Fie', 'Foe', 'Flop')
```

Это означает, что вы можете изменить кортеж следующим образом:

```
>>> t1 = ('Fee', 'Fie', 'Foe')
>>> t2 = ('Flop',)
>>> t1 += t2
>>> t1
('Fee', 'Fie', 'Foe', 'Flop')
```

Это уже не тот же самый кортеж `t1`. Python создал новый кортеж из исходных `t1` и `t2` и присвоил ему имя `t1`. С помощью `id()` вы можете увидеть, когда имя переменной будет указывать на новое значение:

```
>>> t1 = ('Fee', 'Fie', 'Foe')
>>> t2 = ('Flop',)
>>> id(t1)
4365405712
>>> t1 += t2
>>> id(t1)
4364770744
```

Списки

Списки особенно удобны для хранения в них объектов в определенном порядке, особенно если порядок или содержимое нужно будет изменить. В отличие от строк список изменяем: вы можете добавить новые элементы, перезаписать существующие и удалить ненужные. Одно и то же значение может встречаться в списке несколько раз.

Создаем списки с помощью скобок []

Список можно создать из нуля и 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']
>>> leap_years = [2000, 2004, 2008]
>>> randomness = ['Punxsatawney", {"groundhog": "Phil"}, "Feb. 2"}
```

Список `first_names` показывает, что значения не должны быть уникальными.

Если вы хотите размещать в последовательности только уникальные значения и вам неважен их порядок, множество (set) может оказаться более удобным вариантом, чем список. В предыдущем примере список big_birds вполне мог быть множеством. О множествах вы прочитаете в главе 8.

Создаем список или преобразуем в список с помощью функции list()

Вы также можете создать пустой список с помощью функции `list()`:

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

Функция `list()` преобразует другие *итерабельные* типы данных (например, кортежи, строки, множества и словари) в списки. В следующем примере строка преобразуется в список, состоящий из односимвольных строк:

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

В этом примере кортеж преобразуется в список:

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

Создаем список из строки с использованием функции split()

Как я упоминал в разделе «Разделяем строку с помощью функции split()» главы 5, функцию `split()` можно использовать для преобразования строки в список, указав некую строку-разделитель:

```
>>> talk_like_a_pirate_day = '9/19/2019'
>>> talk_like_a_pirate_day.split('/')
['9', '19', '2019']
```

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']
```

Получаем элемент с помощью конструкции [смещение]

Как и в случае со строками, вы можете извлечь одно значение из списка, указав его смещение:

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

Так же и отрицательные индексы отсчитываются с конца строки:

```
>>> marxes[-1]
'Harpo'
>>> marxes[-2]
'Chico'
>>> marxes[-3]
'Groucho'
>>>
```

Смещение должно быть допустимым для этого списка — позицией, которой вы ранее присвоили 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
```

Извлекаем элементы с помощью разделения

Можно извлечь из списка подсписок, использовав *разделение* (slice):

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

Такой фрагмент списка тоже 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']
```

Ни одно из этих разделений не затронуло сам список `marxes`, поскольку мы не выполняли присваивание. Чтобы изменить порядок элементов в списке, используйте функцию `list.reverse()`:

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

Функция reverse() изменяет список, но не возвращает его значения.

Как и в случае со строками, если при разделении указать некорректный индекс, исключение не сгенерируется. Будет использован ближайший корректный индекс или же возвращено пустое значение:

```
>>> marxes = ['Groucho', 'Chico', 'Harpo']
>>> marxes[4:]
[]
>>> marxes[-6:]
['Groucho', 'Chico', 'Harpo']
>>> marxes[-6:-2]
['Groucho']
>>> marxes[-6:-4]
[]
```

Добавляем элемент в конец списка с помощью функции append()

Традиционный способ добавления элементов в список — вызов метода `append()`, который один за одним добавит 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 = ['Groucho', 'Chico', 'Harpo']
>>> marxes.append('Zeppo')
>>> marxes
['Groucho', 'Chico', 'Harpo', 'Zeppo']
```

Добавляем элемент на определенное место с помощью функции insert()

Функция `append()` добавляет элементы только в конец списка. Когда вам нужно добавить элемент и поставить его на заданную позицию, используйте функцию `insert()`. Если вы укажете смещение `0`, элемент будет добавлен в начало списка. Если значение смещения выходит за пределы списка, элемент будет добавлен в ко-

нец, как делает и функция append(): таким образом, вам не нужно беспокоиться о том, что Python сгенерирует исключение:

```
>>> marxes = ['Groucho', 'Chico', 'Harpo']
>>> marxes.insert(2, 'Gummo')
>>> marxes
['Groucho', 'Chico', 'Harpo', 'Gummo']
>>> marxes.insert(10, 'Zeppo')
>>> marxes
['Groucho', 'Chico', 'Harpo', 'Gummo', 'Zeppo']
```

Размножаем элементы с помощью оператора *

В главе 5 вы видели, что можно размножить символы строки с помощью оператора *. Точно так же можно сделать и со списками:

```
>>> ["blah"] * 3
['blah', 'blah', 'blah']
```

Объединяем списки с помощью метода extend() или оператора +

Можно объединить один список с другим, используя extend(). Предположим, что некий 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 операторы + или +=:

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

Если бы мы использовали append(), список others был бы добавлен как *один* из элементов списка, а не дополнил бы своими элементами список marxes:

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

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

Изменяем элемент с помощью конструкции [смещение]

Так же как значение какого-либо элемента из списка можно получить по смещению, его можно и изменить:

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

Опять же смещение должно быть корректным для заданного списка.

Вы не можете таким способом изменить символ в строке, поскольку строки, в отличие от списков, неизменяемы. В списке можно изменить как количество элементов, так и сами элементы.

Изменяем элементы с помощью разделения

В предыдущем разделе вы увидели, как получить подсписок с помощью разделения. Помимо этого, с помощью разделения можно присвоить значения подсписку:

```
>>> numbers = [1, 2, 3, 4]
>>> numbers[1:3] = [8, 9]
>>> numbers
[1, 8, 9, 4]
```

То, что находится справа от = и что вы присваиваете списку, может содержать иное количество элементов, нежели список, указанный слева:

```
>>> numbers = [1, 2, 3, 4]
>>> numbers[1:3] = [7, 8, 9]
>>> numbers
[1, 7, 8, 9, 4]

>>> numbers = [1, 2, 3, 4]
>>> numbers[1:3] = []
>>> numbers
[1, 4]
```

На самом деле то, что находится справа от оператора присваивания, может даже не быть списком. Подойдет любой итерабельный объект, элементы которого можно сделать элементами списка:

```
>>> numbers = [1, 2, 3, 4]
>>> numbers[1:3] = (98, 99, 100)
>>> numbers
[1, 98, 99, 100, 4]

>>> numbers = [1, 2, 3, 4]
>>> numbers[1:3] = 'wat?'
>>> numbers
[1, 'w', 'a', 't', '?', 4]
```

Удаляем заданный элемент с помощью оператора del

Только что наши консультанты проинформировали нас о том, что Гуммо (Gummo) был одним из братьев Маркс, а Карл (Karl) — не был, и кто бы ни добавил его ранее, он поступил очень неосмотрительно. Давайте это исправим:

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

Когда вы удаляете заданный с помощью смещения элемент, все идущие следом за ним элементы сдвигаются, чтобы занять место удаленного, а длина списка уменьшается на единицу. Если вы удалите `Chico` из последней версии списка `marxes`, то получите такой результат:

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

del является оператором Python, а не методом списка — нельзя написать marxes[–1].del(). Это своего рода обратное присваивание, так как del выполняет противоположную присваиванию (=) операцию: открепляет имя от объекта Python и может освободить место объекта в памяти, если это имя являлось последней ссылкой на него.

Удаляем элемент 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, Groucho:

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

Если у вас были повторяющиеся элементы списка с одинаковым значением, функция `remove()` удалит только первый найденный элемент.

Получаем и удаляем заданный элемент с помощью функции pop()

Получить элемент из списка и так же удалить его можно с помощью функции `pop()`. Если вызвать функцию `pop()` и указать смещение, она возвратит элемент, находящийся в заданной позиции. Если аргумент не указать, будет использовано

значение -1. Так, вызов pop(0) вернет начальный элемент списка, а вызов pop() или pop(-1) — конечный, как показано далее:

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

Пришло время для 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 что-либо из конца того же списка, вы работаете со структурой данных, известной как очередь LIFO (last in, first out — «последним пришел, первым ушел»). Ее чаще называют стеком. Вызов pop(0) создает очередь FIFO (first in, first out — «первым пришел, первым ушел»). Это удобный способ собирать данные по мере их поступления и работать либо с самыми старыми (FIFO), либо с самыми новыми (LIFO).

Удаляем все элементы с помощью функции clear()

В Python 3.3 появился метод, позволяющий удалить все элементы из списка:

```
>>> work_quotes = ['Working hard?', 'Quick question!', 'Number one priorities!']
>>> work_quotes
['Working hard?', 'Quick question!', 'Number one priorities!']
>>> work_quotes.clear()
>>> work_quotes
[]
```

Определяем смещение 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
```

Если значение встречается в списке более одного раза, возвращается смещение только первого найденного элемента:

```
>>> simpsons = ['Lisa', 'Bart', 'Marge', 'Homer', 'Bart']
>>> simpsons.index('Bart')
1
```

Проверяем 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 находиться в списке на нескольких позициях. Пока оно присутствует 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 вам неважен порядок элементов, то для хранения и поиска уникальных значений больше подойдет множество. О множествах мы поговорим чуть позже в главе 8.

Подсчитываем количество включений значения с помощью функции count()

Чтобы подсчитать, 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
```

Преобразуем список в строку с помощью функции join()

В разделе «Объединяем строки с помощью функции join()» главы 5 функция `join()` рассматривается более подробно, но вот еще один пример того, что можно сделать с ее помощью:

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

На первый взгляд это выглядит неправильным. Ведь функция `join()` предназначается для строк, а не для списков. Вы не можете написать `marxes.join(', ')`, даже если интуиция вам так подсказывает. Аргументом для функции `join()` является строка или любая итерабельная последовательность строк (в том числе и список), а выводом — строка. Если бы функция `join()` была только методом списка, ее нельзя было бы использовать для других итерабельных объектов, таких как кортежи и строки. Если вы хотите, чтобы она работала с любым итерабельным типом, нужно написать особый код для каждого типа и таким образом обработать подобное объединение. Это поможет вам 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 функций sort() или sorted()

Часто вам будет нужно сортировать элементы в списке по значению, а не по смещению. 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 строками — 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` список изменит:

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

Если все элементы списка одного типа (в списке `marxes` находятся только строки), функция `sort()` отработает корректно. Иногда можно даже смешать типы, на-

пример, целые числа и числа с плавающей точкой, поскольку в рамках выражений они автоматически преобразуются друг в друга:

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

По умолчанию список сортируется по возрастанию, но можно 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]
```

Получаем длину списка с помощью функции len()

Функция `len()` возвращает количество элементов списка:

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

Присваиваем с помощью оператора =

Когда вы присваиваете один список более чем одной переменной, изменение списка в одном месте также меняет его и в других, 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 собой представляет переменная `b`? Ее значение все еще равно `[1, 2, 3]` или изменилось на `['surprise', 2, 3]`? Проверим:

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

Помните аналогию с коробкой (объектом) и этикеткой (именем переменной) из главы 2? `b` просто ссылается на тот же список объектов, что и `a` (обе этикетки указывают на одну и ту же коробку с объектом), поэтому, независимо от того, с помощью какого имени мы изменяем содержимое списка, изменение отражается на обеих переменных:

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

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

Копируем списки с помощью функций copy() и list() или путем разделения

Вы можете скопировать значения в отдельный новый список с помощью любого из этих методов:

- функции `copy()`;
- функции преобразования `list()`;
- разделения 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 разделения списка:

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

Опять же `b`, `c` и `d` являются *копиями* `a` — новыми объектами со своими собственными значениями и не связанными с исходным списком объектов `[1, 2, 3]`, на который ссылается `a`. Изменение `a` *не* влияет 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]
```

Копируем все с помощью функции deepcopy()

Функция `copy()` хорошо работает, если все элементы списка являются неизменяемыми. Как вы видели ранее, изменяемые значения (например, списки, кортежи или словари) являются ссылками. Изменение оригинала или копии отразится на каждом из них.

Воспользуемся предыдущим примером, но в качестве последнего элемента списка `a` вместо целого числа `3` используем список `[8, 9]`:

```
>>> a = [1, 2, [8, 9]]
>>> b = a.copy()
>>> c = list(a)
>>> d = a[:]
```

```
>>> a
[1, 2, [8, 9]]
>>> b
[1, 2, [8, 9]]
>>> c
[1, 2, [8, 9]]
>>> d
[1, 2, [8, 9]]
```

Пока все хорошо. Теперь изменим элемент нашего подсписка из списка `a`:

```
>>> a[2][1] = 10
>>> a
[1, 2, [8, 10]]
>>> b
[1, 2, [8, 10]]
>>> c
[1, 2, [8, 10]]
>>> d
[1, 2, [8, 10]]
```

Значение элемента `a[2]` теперь является списком, и его элементы могут быть изменены. Все методы копирования списков, которые мы уже рассмотрели, были *поверхностными*: это не оценочное суждение — речь идет именно о том, насколько глубоко распространяется копирование.

Для того чтобы это исправить, нужно использовать функцию `deepcopy()`:

```
>>> import copy
>>> a = [1, 2, [8, 9]]
>>> b = copy.deepcopy(a)
>>> a
[1, 2, [8, 9]]
>>> b
[1, 2, [8, 9]]
>>> a[2][1] = 10
>>> a
[1, 2, [8, 10]]
>>> b
[1, 2, [8, 9]]
```

Функция `deepcopy()` может работать с вложенными списками, словарями и другими объектами.

Подробнее о директиве `import` вы узнаете из главы 11.

Сравниваем списки

Сравнить списки можно с помощью операторов сравнения, таких как `==`, `<` и другие. Операторы проходят по обоим спискам, сравнивая элементы с одинаковым смещением. Если список `a` короче списка `b` и все элементы списка `a` присутствуют в списке `b`, будет считаться, что список `a` меньше списка `b`:

```
>>> a = [7, 2]
>>> b = [7, 2, 9]
```

```
>>> a == b
False
>>> a <= b
True
>>> a < b
True
```

Итерируем по спискам с помощью операторов for и in

Из главы 6 вы узнали, как можно проитерировать по строке с помощью цикла `for`. Однако гораздо чаще доводится выполнять итерирование по спискам:

```
>>> cheeses = ['brie', 'gjetost', 'havarti']
>>> for cheese in cheeses:
...     print(cheese)
...
brie
gjetost
havarti
```

Как и раньше, ключевое слово `break` завершает выполнение цикла `for`, а ключевое слово `continue` позволяет перейти к следующей итерации:

```
>>> cheeses = ['brie', 'gjetost', 'havarti']
>>> for cheese in cheeses:
...     if cheese.startswith('g'):
...         print("I won't eat anything that starts with 'g'")
...         break
...     else:
...         print(cheese)
...
brie
I won't eat anything that starts with 'g'
```

Вы по-прежнему можете использовать опциональный блок `else`, если цикл `for` завершился без `break`:

```
>>> cheeses = ['brie', 'gjetost', 'havarti']
>>> for cheese in cheeses:
...     if cheese.startswith('x'):
...         print("I won't eat anything that starts with 'x'")
...         break
...     else:
...         print(cheese)
... else:
...     print("Didn't find anything that started with 'x'")
...
brie
gjetost
havarti
Didn't find anything that started with 'x'
```

Если цикл `for` не отработал ни разу, выполнение также продолжится в блоке `else`:

```
>>> cheeses = []
>>> for cheese in cheeses:
...     print('This shop has some lovely', cheese)
...     break
... else:  # отсутствие break означает отсутствие сыра
...     print('This is not much of a cheese shop, is it?')
...
This is not much of a cheese shop, is it?
```

Поскольку список `cheeses` в этом примере был пуст, цикл `for cheese in cheeses` ни разу не отработал, а оператор `break` не был применен.

Итерируем по нескольким последовательностям с помощью функции zip()

Есть еще один хороший 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()` прекратит свою работу, когда будет выполнена самая короткая последовательность. Один из списков (`desserts`) оказался длиннее остальных, поэтому никто не получит пудинг, пока мы не увеличим остальные списки.

В главе 8 показывается, как с помощью функции `dict()` можно создавать словари из двухэлементных последовательностей, таких как кортежи, списки или строки. Функция `zip()` позволяет пройти по нескольким последовательностям и создать кортежи из элементов с одинаковым смещением. 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 с составлением пар. 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')]
```

Передадим 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'}
```

Создаем список с помощью списковых включений

Вы уже знаете, что список можно создать с помощью квадратных скобок и функции `list()`. Сейчас мы рассмотрим создание списка с помощью *списковых включений* (их еще называют генераторами списков, представлением списков и т. д.): этот механизм использует итерирование с помощью ключевых слов `for`/`in`, которое мы только что рассмотрели.

Вы можете создать список целых чисел от 1 до 5, добавляя их по одному, 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 используя 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 преобразуя в список сам результат работы функции `range()`:

```
>>> number_list = list(range(1, 6))
>>> number_list
[1, 2, 3, 4, 5]
```

Все эти приемы абсолютно корректны с точки зрения Python и приводят к одинаковому результату. Однако более характерным для Python является создание списка с помощью *списковых включений*. Простейшая форма такого включения выглядит следующим образом:

```
[выражение for элемент in итерабельный объект]
```

Вот так списковое включение может составить список целых чисел:

```
>>> number_list = [number for number in range(1,6)]
>>> number_list
[1, 2, 3, 4, 5]
```

Первая переменная `number` нужна для формирования значений для списка, то есть для размещения результата работы цикла в переменной `number_list`. Вторая пере-

менная 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]
```

Списковое включение перемещает цикл в квадратные скобки. Этот пример включения не намного проще предыдущего, однако можно пойти дальше и добавить условное выражение, которое выглядит следующим образом:

```
[выражение for элемент in итерабельный объект if условие]
```

Используем новое включение для создания списка 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]
```

Такое включение выглядит чуть более компактно, чем в обычном варианте:

```
>>> 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 при работе с вложенными циклами, можно написать более чем один набор операторов for ... в соответствующем включении. 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 получить 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 ...` в списковом включении также могут иметь свои проверки `if`.

Списки списков

Списки могут содержать элементы различных типов, в том числе и другие списки, как 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']]
```

Посмотрим на его первый элемент:

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

Первый элемент является списком — это список `small_birds`, который мы указали как первый элемент списка `all_birds`. Нетрудно догадаться, чем является второй элемент:

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

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

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

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

Кортежи или списки?

Вместо списков можно использовать кортежи, однако у них меньше возможностей — нет функций `append()`, `insert()` и т. д., поскольку кортеж не может быть изменен после создания. Почему бы тогда не работать всегда со списками вместо кортежей?

- Кортежи занимают меньше места.
- Вы не сможете уничтожить элементы кортежа по ошибке.
- Вы можете использовать кортежи в качестве словарных ключей (см. главу 8).
- *Именованные кортежи* (см. раздел «Именованные кортежи» в главе 10) могут служить более простой альтернативой объектам.

Здесь я не буду вдаваться в подробности использования кортежей. Чаще при решении повседневных задач вам доведется обращаться именно к спискам и словарям.

Включений кортежей не существует

Изменяемые типы (списки, словари и множества) имеют включения. Неизменяемые типы, такие как строки и кортежи, нужно создавать другими методами, описанными в соответствующих разделах.

Вы можете подумать, что замена квадратных скобок спискового включения круглыми скобками создаст кортежное включение. Может даже показаться, что это работает, поскольку исключение не сгенерируется:

```
>>> number_thing = (number for number in range(1, 6))
```

Однако в круглых скобках находится что-то совершенно иное — включение генератора, и оно вернет *объект генератора*:

```
>>> type(number_thing)
<class 'generator'>
```

Генераторы мы рассмотрим более подробно в разделе «Генераторы» главы 9. Генератор — это один из способов предоставления данных итератору.

Читайте далее

Следующие темы — *словари* и *множества* — настолько объемные, что заслужили отдельную главу.

Упражнения

Используйте списки и кортежи из чисел (глава 3) и строки (глава 3), чтобы представить самые разные элементы реального мира.

7.1. 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, но вы уже читаете эту книгу, то я даже не знаю, что сказать.

7.2. В какой год из списка `years_list` был ваш третий день рождения? Учтите, в первый год вам было 0 лет.

7.3. В какой год из списка `years_list` вам было больше всего лет?

7.4. Создайте список `things`, содержащий три элемента: `"mozzarella"`, `"cinderella"`, `"salmonella"`.

7.5. Напишите с большой буквы тот элемент списка `things`, который означает человека, а затем выведите список. Изменился ли элемент списка?

7.6. Переведите сырный элемент списка `things` в верхний регистр целиком и выведите список.

7.7. Удалите из списка `things` заболевание, получите Нобелевскую премию и затем выведите список на экран.

7.8. Создайте список с элементами `"Groucho"`, `"Chico"` и `"Harpo"`; назовите его `surprise`.

7.9. Напишите последний элемент списка `surprise` со строчной буквы, затем выведите эту строку в обратном порядке и с прописной буквы.

7.10. Используйте списковое включение, чтобы создать список с именем `even`, в котором будут содержаться четные числа в промежутке `range(10)`.

7.11. Попробуйте создать генератор рифм для прыжков через скакалку. Напечатайте последовательность двухстрочных рифм. Начните программу с этого фрагмента:

```
start1 = ["fee", "fie", "foe"]
rhymes = [
    ("flop", "get a mop"),
    ("fope", "turn the rope"),
    ("fa", "get your ma"),
    ("fudge", "call the judge"),
    ("fat", "pet the cat"),
```

```
    ("fog", "walk the dog"),
    ("fun", "say we're done"),
    ]
start2 = "Someone better"
```

Затем следуйте инструкциям.

Для каждого кортежа `(first, second)` в списке `rhymes`.

- Для первой строки:
  - выведите на экран каждую строку списка `start1`: начните ее с большой буквы, а закончите восклицательным знаком с пробелом;
  - выведите на экран значение переменной `first`, также записав его с большой буквы и с восклицательным знаком на конце.
- Для второй строки:
  - выведите на экран значение переменной `start2` и пробел;
  - выведите на экран значение переменной `second` и точку.

ГЛАВА 8
Словари и множества

> Если в словаре слово записано неправильно, как нам об этом узнать?
>
> *Стивен Райт*

Словари

Словарь очень похож на список, но порядок элементов в нем не имеет значения и они не выбираются смещением, таким как 0 и 1. Вместо этого для каждого *значения* вы указываете связанный с ним уникальный *ключ*. Этот ключ чаще всего представляет собой строку, но, в принципе, может быть любым из неизменяемых типов: булевой переменной, целым числом, числом с плавающей точкой, кортежем, строкой, а также другими объектами, с которыми вы познакомитесь далее. Словари являются изменяемыми, а это значит, что вы можете добавить, удалить и изменить их элементы, имеющие вид «ключ — значение».

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

В других языках словари могут называться *ассоциативными массивами, хешами* или *хеш-таблицей*. В Python словарь также принято называть dict.

Создаем словарь с помощью {}

Чтобы создать словарь, нужно заключить 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
{'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'}
```

В Python допускается наличие запятой после последнего элемента списка, кортежа или словаря. Кроме того, не обязательно делать отступы, как делал я в предыдущем примере; когда вы вводите ключи и значения внутри фигурных скобок — это просто улучшает читабельность.

Создаем словарь с помощью функции dict()

Некоторым людям не нравится печатать так много фигурных скобок и кавычек. И вы можете создать словарь, передав именованные аргументы и значения в функцию `dict()`.

Традиционный способ создания словаря:

```
>>> acme_customer = {'first': 'Wile', 'middle': 'E', 'last': 'Coyote'}
>>> acme_customer
{'first': 'Wile', 'middle': 'E', 'last': 'Coyote'}
```

С использованием `dict()`:

```
>>> acme_customer = dict(first="Wile", middle="E", last="Coyote")
>>> acme_customer
{'first': 'Wile', 'middle': 'E', 'last': 'Coyote'}
```

Одним из ограничений второго способа является то, что имена аргументов должны представлять собой корректные имена переменных (в них не должны использоваться пробелы и ключевые слова):

```
>>> x = dict(name="Elmer", def="hunter")
  File "<stdin>", line 1
    x = dict(name="Elmer", def="hunter")
                           ^
SyntaxError: invalid syntax
```

Преобразуем с помощью функции dict()

С помощью функции `dict()` можно преобразовывать двузначные последовательности в словари. (Иногда вы можете столкнуться с такими последовательностями «ключ — значение», как «Стронций, 90, углерод, 14»[1].) Первый элемент каждой последовательности применяется как ключ, второй — как значение.

Для начала рассмотрим небольшой пример использования `lol` — списка, состоящего из двузначных списков:

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

Мы могли бы использовать любую последовательность, содержащую последовательности из двух элементов. Рассмотрим другие 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 «Итерируем по нескольким последовательностям с помощью функции zip()» главы 7 вы уже познакомились с функцией `zip()`, которая позволит вам легко создавать такие двухэлементные последовательности.

Добавляем или изменяем элемент с помощью конструкции [ключ]

Добавить элемент в словарь довольно легко. Просто обратитесь к элементу по его ключу и присвойте ему значение. Если ключ уже существует в словаре, имеющееся значение будет заменено новым. Если ключ новый, он и указанное значение будут добавлены в словарь. В отличие от списков, здесь вам не нужно беспокоиться о том,

[1] Очень похоже на финальный счет матча «Стронций» — «Углерод».

что Python сгенерирует исключение во время присваивания нового элемента, если вы укажете индекс, который выходит за пределы существующего диапазона.

Сейчас мы создадим словарь с перечислением участников комик-группы из Великобритании 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
{'Chapman': 'Graham', 'Cleese': 'John', 'Idle': 'Eric',
'Jones': 'Terry', 'Palin': 'Michael'}
```

Не хватает одного участника — уроженца Америки Терри Гиллиама. Здесь вы увидите попытку неизвестного программиста его добавить. Однако программист ошибся, когда вводил имя:

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

А вот код другого программиста, который исправил эту ошибку:

```
>>> pythons['Gilliam'] = 'Terry'
>>> pythons
{'Chapman': 'Graham', 'Cleese': 'John', 'Idle': 'Eric',
'Jones': 'Terry', 'Palin': 'Michael', 'Gilliam': '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 примените ключ более одного раза, победит последнее значение:

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

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

Получаем элемент словаря с помощью конструкции [ключ] или функции get()

Этот вариант использования словаря — самый распространенный. Вы указываете словарь и ключ, чтобы получить соответствующее значение. Применим `some_phyton` из предыдущего подраздела:

```
>>> some_pythons['John']
'Cleese'
```

Если ключа в словаре нет, будет сгенерировано исключение:

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

Есть два хороших способа избежать возникновения подобного исключения. Первый — с помощью ключевого слова `in` проверить, имеется ли заданный ключ. Этот способ вы уже видели в предыдущем подразделе:

```
>>> 'Groucho' 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('John')
'Cleese'
```

Если такого ключа нет, вы получите опциональное значение:

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

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

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

Получаем все ключи с помощью функции keys()

Вы можете использовать функцию `keys()`, чтобы получить все ключи словаря. Для следующих примеров обратимся к другому словарю:

```
>>> 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 требуется 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 необходимо применять функцию `list()`, чтобы преобразовывать результат работы функций `values()` и `items()` в обычные списки. Я пользуюсь этой функцией в своих примерах.

Получаем все значения с помощью функции values()

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

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

Получаем все пары «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 в виде кортежа, например `('green', 'go')`.

Получаем длину словаря с помощью функции len()

Количество пар «ключ — значение» определяется так:

```
>>> len(signals)
3
```

Объединяем словари с помощью конструкции {**a, **b}

В версиях Python 3.5 и выше есть новый способ объединять словари с помощью конструкции **, которая в главе 9 применяется совершенно по-иному:

```
>>> first = {'a': 'agony', 'b': 'bliss'}
>>> second = {'b': 'bagels', 'c': 'candy'}
>>> {**first, **second}
{'a': 'agony', 'b': 'bagels', 'c': 'candy'}
```

Фактически вы можете передать в качестве параметров больше двух словарей:

```
>>> third = {'d': 'donuts'}
>>> {**first, **third, **second}
{'a': 'agony', 'b': 'bagels', 'd': 'donuts', 'c': 'candy'}
```

Эти копии являются *поверхностными*. Обратитесь к подразделу, в котором рассматривается функция `deepcopy()` (см. подраздел «Копируем все с помощью функции deepcopy()» далее в этой главе), если вам нужно получить полные копии ключей и значений, не связанные с исходными словарями.

Объединяем словари 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
{'Chapman': 'Graham', 'Cleese': 'John', 'Gilliam': 'Terry',
'Idle': 'Eric', 'Jones': 'Terry', 'Palin': 'Michael'}
```

Кроме того, у нас есть еще один словарь — `others`, содержащий имена других юмористов:

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

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

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

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
{'a': 1, 'b': 'platypus'}
```

Удаляем элементы по их ключу с помощью оператора del

Код нашего анонимного программиста `pythons.update(others)` был корректным технически, но не фактически. Члены словаря `others`, несмотря на свою известность и чувство юмора, не участвовали в Monty Python. Отменим добавление последних двух элементов:

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

Получаем элемент по ключу и удаляем его с помощью функции pop()

В этой функции объединены функции `get()` и `del`. Если вы передадите функции `pop()` ключ в качестве аргумента и такой ключ имеется в словаре, она вернет соответствующее значение и удалит пару. Если ключа в словаре нет, будет сгенерировано исключение:

```
>>> len(pythons)
6
>>> pythons.pop('Palin')
'Michael'
>>> len(pythons)
5
>>> pythons.pop('Palin')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'Palin'
```

Но если вы передадите функции `pop()` второй аргумент по умолчанию (как и в случае с функцией `get()`), словарь не изменится:

```
>>> pythons.pop('First', 'Hugo')
'Hugo'
>>> len(pythons)
5
```

Удаляем все элементы с помощью функции clear()

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

```
>>> pythons.clear()
>>> pythons
```

```
{}
>>> pythons = {}
>>> pythons
{}
```

Проверяем на наличие ключа с помощью оператора in

Если вы хотите узнать, содержится ли в словаре определенный ключ, используйте ключевое слово `in`. Снова переопределим словарь `pythons`, но на этот раз пропустим имена одного-двух участников:

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

Теперь проверим, кого мы добавили:

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

На этот раз мы не забыли о Терри Гиллиаме?

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

Черт!

Присваиваем значения с помощью оператора =

Как и в случае со списками, если вы внесете в словарь изменение, оно отразится на всех именах, которые на него ссылаются:

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

Копируем значения с помощью функции copy()

Чтобы скопировать ключи и значения из одного словаря в другой и запретить изменяться остальным словарям при изменении данных в одном из них, можно воспользоваться функцией `copy()`:

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

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

Эта копия является *поверхностной*: ее можно применять тогда, когда значения словаря неизменяемы (как в нашем случае). Если значения словаря изменяются, следует использовать функцию `deepcopy()`.

Копируем все с помощью функции deepcopy()

Если бы значением по ключу `red` в предыдущем примере был список, а не обычная строка:

```
>>> signals = {'green': 'go',
... 'yellow': 'go faster',
... 'red': ['stop', 'smile']}
>>> signals_copy = signals.copy()
>>> signals
{'green': 'go',
'yellow': 'go faster',
'red': ['stop', 'smile']}
>>> signals_copy
{'green': 'go',
'yellow': 'go faster',
'red': ['stop', 'smile']}
>>>
```

Изменим одно из значений в списке `red`:

```
>>> signals['red'][1] = 'sweat'
>>> signals
{'green': 'go',
'yellow': 'go faster',
'red': ['stop', 'sweat']}
>>> signals_copy
{'green': 'go',
'yellow': 'go faster',
'red': ['stop', 'sweat']}
```

Вы видите типичную картину, когда при внесении изменений в один объект изменяются оба объекта. Метод `copy()` копировал значения как есть — это значит, что `signal_copy` получил тот же список значений для `'red'`, которые имел и `signals`.

Решением проблемы является функция `deepcopy()`:

```
>>> import copy
>>> signals = {'green': 'go',
```

```
... 'yellow': 'go faster',
... 'red': ['stop', 'smile']}
>>> signals_copy = copy.deepcopy(signals)
>>> signals
{'green': 'go',
'yellow': 'go faster',
'red': ['stop', 'smile']}
>>> signals_copy
{'green': 'go',
'yellow':'go faster',
'red': ['stop', 'smile']}
>>> signals['red'][1] = 'sweat'
>>> signals
{'green': 'go',
'yellow': 'go faster',
red': ['stop', 'sweat']}
>>> signals_copy
{'green': 'go',
'yellow': 'go faster',
red': ['stop', 'smile']}
```

Сравниваем словари

Как и в случае со списками и кортежами, рассмотренными в предыдущей главе, словари можно сравнивать с помощью операторов `==` и `!=`:

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

Другие операторы не будут работать:

```
>>> a = {1:1, 2:2, 3:3}
>>> b = {3:3, 1:1, 2:2}
>>> a <= b
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: '<=' not supported between instances of 'dict' and 'dict'
```

Python сравнивает ключи и значения по одному. Порядок, в котором они создавались, не имеет значения. В этом примере словари `a` и `b` равны, за исключением того, что по ключу `1` в словаре `a` находится список `[1, 2]`, а в словаре `b` — список `[1, 1]`.

```
>>> a = {1: [1, 2], 2: [1], 3:[1]}
>>> b = {1: [1, 1], 2: [1], 3:[1]}
>>> a == b
False
```

Итерируем по словарям с помощью for и in

Итерирование по словарю (или его функция `keys()`) возвращает ключи. В этом примере в качестве ключей используются типы карт для настольной игры Clue (за пределами Северной Америки она называется Cluedo):

```
>>> accusation = {'room': 'ballroom', 'weapon': 'lead pipe',
...               'person': 'Col. Mustard'}
>>> for card in accusation:  #  или for card in accusation.keys():
...     print(card)
...
room
weapon
person
```

Для того чтобы проитерировать не по ключам, а по значениям, используйте функцию `values()`:

```
>>> for value in accusation.values():
...     print(value)
...
ballroom
lead pipe
Col. Mustard
```

Для получения пар «ключ — значение» подходит функция `items()`:

```
>>> for item in accusation.items():
...     print(item)
...
('room', 'ballroom')
('weapon', 'lead pipe')
('person', 'Col. Mustard')
```

Присвоить значение кортежа другим переменным можно в один этап. Для каждого кортежа, возвращенного функцией `items()`, присвоим первый элемент кортежа (ключ) переменной `card`, а второй элемент (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
```

Включения словарей

У словарей также существуют включения.

Простейшая форма выглядит знакомо:

```
{выражение для ключа : выражение для значения for выражение in итерабельный объект}
>>> 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 проходим в цикле по каждой из семи букв строки `'letters'` и считаем, сколько раз буква встречается. Дважды использовать функцию `word.count(letter)` — значит впустую тратить время, поскольку нам придется по два раза посчитать буквы `e` и `t`. Но это ничему не навредит, потому что после второго вызова этой функции для буквы `e` мы всего лишь заменим существующую запись в словаре.

То же самое верно и для буквы `t`. Следующий фрагмент больше соответствует «питонскому» стилю:

```
>>> 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}
```

Порядок ключей словаря изменился по сравнению с предыдущим примером, поскольку вызов `set(word)` возвращает буквы в другом порядке.

По аналогии со списковыми включениями для генератора словарей также можно использовать условные проверки `if` и более одного блока `for`:

```
{выражение для ключа : выражение для значения for выражение in итерабельный
объект if условие}
```

```
>>> vowels = 'aeiou'
>>> word = 'onomatopoeia'
>>> vowel_counts = {letter: word.count(letter) for letter in set(word)
        if letter in vowels}
>>> vowel_counts
{'e': 1, 'i': 1, 'o': 4, 'a': 2}
```

Вы сможете найти дополнительные примеры использования генераторов словарей в PEP-274 (https://oreil.ly/6udkb).

Множества

Множество похоже на словарь, в котором значения отброшены, а оставлены только ключи. Как и в словаре, ключи должны быть уникальными. Вы используете множество, если хотите знать только, существует что-то или нет, а другая информация вам неважна. Это как сумка с ключами. Если же вам нужно прикрепить к ключу некую информацию, воспользуйтесь словарем.

Было время, когда теорию множеств преподавали в некоторых начальных школах наряду с основами математики. Если в вашей школе такого не было (или было, но вы в это время смотрели в окно), на рис. 8.1 можно увидеть, как объединяются и пересекаются множества.

Предположим, вы хотите объединить два множества, у которых есть несколько общих ключей. Поскольку множество должно содержать только уникальные значения, объединение двух множеств будет иметь всего лишь один такой одинаковый ключ. *Пустое* множество — это

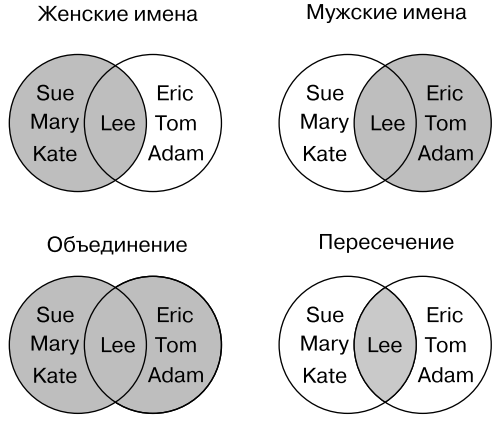

Рис. 8.1. Обычные операции с множествами

множество, содержащее ноль элементов. На рис. 8.1 примером пустого множества будут женские имена, начинающиеся с буквы X.

Создаем множество с помощью функции set()

Чтобы создать множество, воспользуйтесь функцией `set()`. Можно также 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, 2, 4, 6, 8}
>>> odd_numbers = {1, 3, 5, 7, 9}
>>> odd_numbers
{1, 3, 5, 7, 9}
```

Порядок ключей в множестве не имеет значения.

Поскольку пустые квадратные скобки [] создают пустой список, можно было бы предположить, 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. А владение — это девять десятых законного права.

Преобразуем другие типы данных с помощью функции set()

Вы можете создать множество из списка, строки, кортежа или словаря, отбрасывая любые повторяющиеся значения.

Для начала взглянем на строку, в которой отдельные буквы встречаются несколько раз:

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

Обратите внимание, что множество содержит только по одной `e` и `t`, хотя в слове `letters` они встречаются дважды.

Создадим множество из списка:

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

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'})
{'cherry', 'orange', 'apple'}
```

Получаем длину множества с помощью функции len()

Пересчитаем наших оленей:

```
>>> reindeer = set( ['Dasher', 'Dancer', 'Prancer', 'Mason-Dixon'] )
>>> len(reindeer)
4
```

Добавляем элемент с помощью функции add()

Добавим во множество еще один элемент с помощью метода `add()`:

```
>>> s = set((1,2,3))
>>> s
{1, 2, 3}
>>> s.add(4)
>>> s
{1, 2, 3, 4}
```

Удаляем элемент с помощью функции remove()

Вы можете удалить выбранное значение из множества:

```
>>> s = set((1,2,3))
>>> s.remove(3)
>>> s
{1, 2}
```

Итерируем по множествам с помощью for и in

Как и в случае со словарями, вы можете проитерировать по всем элементам множества:

```
>>> furniture = set(('sofa', 'ottoman', 'table'))
>>> for piece in furniture:
...     print(piece)
...
ottoman
table
sofa
```

Проверяем на наличие значения с помощью оператора in

Такое использование множеств — самое распространенное. Мы создадим словарь, который называется `drinks`. Каждый ключ будет названием коктейля, а соответствующее значение — множеством с ингредиентами этого напитка:

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

Хотя и словарь, и множества заключаются в фигурные скобки (`{` и `}`), множество — это всего лишь набор значений, а словарь содержит пары *«ключ — значение»*.

Какой из коктейлей содержит водку?

```
>>> 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 как 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 кратко в следующем подразделе.

Комбинации и операторы

Что, если вам нужно проверить наличие сразу нескольких значений множества? Предположим, вы хотите найти какой-нибудь напиток с апельсиновым соком или вермутом. 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 со всеми элементами из обоих сравниваемых списков. 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 подраздела, когда 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 не пришлось слишком много печатать в дальнейших примерах:

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

Ниже приведены примеры всех операторов множеств. У одних есть специальная пунктуация, у других — специальные функции, у третьих — и то и другое. Мы будем использовать тестовые множества `a` (содержит элементы `1` и `2`) и `b` (содержит элементы `2` и `3`):

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

Как вы видели ранее, *пересечение* (элементы, общие для обоих множеств) можно получить с помощью особого пунктуационного символа `&`. Функция `intersection()` делает то же самое:

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

В данном фрагменте используются сохраненные нами переменные:

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

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

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

И 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 подраздел и остальные операции, но, возможно, вам никогда не придется 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'}
```

Проверить, является ли одно множество *подмножеством* другого (когда все члены первого множества являются членами второго), можно 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[1].

```
>>> 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 <, как показано в следующем примере:

```
>>> a < b
False
>>> a < a
False

>>> bruss < wruss
True
```

[1] Однако, перефразируя Граучо Маркса, «я бы не хотел состоять в клубе, в который принимают таких людей, как я».

Надмножество 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 *надмножеством самого себя*:

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

И наконец, вы можете встретить *правильное надмножество* (первое множество содержит все члены второго и несколько других) с помощью оператора `>`, как показано здесь:

```
>>> a > b
False

>>> wruss > bruss
True
```

Множество не может быть правильным надмножеством самого себя:

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

Включение множества

Никто не хочет оставаться в стороне, 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}
```

Здесь также могут быть необязательные проверки условий:

```
{ выражение for выражение in итерабельный объект if условие }

>>> a_set = {number for number in range(1,6) if number % 3 == 1}
>>> a_set
{1, 4}
```

Создаем неизменяемое множество с помощью функции frozenset()

Если вы хотите создать множество, которое нельзя будет изменить, вызовите функцию `frozenset()`, передав в нее любой итерабельный аргумент:

```
>>> frozenset([3, 2, 1])
frozenset({1, 2, 3})
>>> frozenset(set([2, 1, 3]))
frozenset({1, 2, 3})
>>> frozenset({3, 1, 2})
frozenset({1, 2, 3})
>>> frozenset( (2, 3, 1) )
frozenset({1, 2, 3})
```

Достаточно ли «заморозилось» наше множество?

```
>>> fs = frozenset([3, 2, 1])
>>> fs
frozenset({1, 2, 3})
>>> fs.add(4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'frozenset' object has no attribute 'add'
```

Да, множество хорошо «заморозилось».

Структуры данных, которые мы уже рассмотрели

Напомню, структуры данных *создаются* следующим образом:

- список — с помощью квадратных скобок ([]);
- кортеж — с помощью запятых (и опциональных скобок);
- словарь и множество — с помощью фигурных скобок ({}).

Во всех случаях, за исключением множеств, вы *получаете доступ* к отдельному элементу с помощью квадратных скобок. Для списка и кортежа значение, находящееся в квадратных скобках, является целочисленным смещением. Для словаря же оно является ключом. Везде результатом будет значение. Для множества: значение либо есть, либо нет; ключей и индексов нет.

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

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 свои более крупные и сложные структуры. Начнем с трех разных списков:

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

Мы можем создать кортеж, элементами которого станут все эти списки:

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

Можем также создать список, состоящий из трех списков:

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

Словарь из списков также возможен. В этом примере используем название группы комиков в качестве ключа, а список ее участников — в качестве значения:

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

Ваши единственные ограничения — только сами типы данных. Например, ключи словаря должны быть неизменяемыми, поэтому списки, словари и множества не могут быть ключами для другого словаря. Но кортеж может быть. Создадим, к примеру, алфавитный указатель достопримечательностей, основываясь на GPS-координатах (широте, долготе и высоте). В главе 21 вы найдете еще несколько примеров работы с картами:

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

Читайте далее

Мы вернемся к структурам кода. Покажем, как «завернуть» код в функцию и как работать с *исключениями*, когда что-то идет не так.

Упражнения

8.1. Создайте англо-французский словарь с названием `e2f` и выведите его на экран. Вот ваши первые слова: `dog/chien`, `cat/chat` и `walrus/morse`.

8.2. Используя словарь `e2f`, выведите французский вариант слова `walrus`.

8.3. Создайте французско-английский словарь `f2e` на основе словаря `e2f`. Используйте метод `items`.

8.4. Используя словарь `f2e`, выведите английский вариант слова `chien`.

8.5. Выведите на экран множество английских слов из ключей словаря `e2f`.

8.6. 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.

8.7. Выведите на экран высокоуровневые ключи словаря `life`.

8.8. Выведите на экран ключи `life['animals']`.

8.9. Выведите значения `life['animals']['cats']`.

8.10. Используйте генератор словаря, чтобы создать словарь `squares`. Используйте `range(10)`, чтобы получить ключи. В качестве значений используйте возведенное в квадрат значение каждого ключа.

8.11. Используйте генератор множества, чтобы создать множество `odd` из нечетных чисел диапазона `range(10)`.

8.12. Используйте включение генератора, чтобы вернуть строку `'Got '` и числа из диапазона `range(10)`. Итерируйте по этой конструкции с помощью цикла `for`.

8.13. Используйте функцию `zip()`, чтобы создать словарь из кортежа ключей (`'optimist'`, `'pessimist'`, `'troll'`) и кортежа значений (`'The glass is half full'`, `'The glass is half empty'`, `'How did you get a glass?'`).

8.14. Используйте функцию `zip()`, чтобы создать словарь с именем `movies`, в котором объединены списки `titles = ['Creature of Habit', 'Crewel Fate', 'Sharks On a Plane']` и `plots = ['A nun turns into a monster', 'A haunted yarn shop', 'Check your exits']`.

ГЛАВА 9
Функции

Чем меньше функция, тем лучше управление.

Сирил Норткот Паркинсон

До этого момента все наши примеры кода были небольшими фрагментами. Они подходят для решения небольших задач, но никому не хочется раз за разом эти фрагменты перепечатывать. Нужна какая-то форма преобразования большого кода в ряд удобных в применении частей.

Первый шаг к повторному использованию кода — создание *функций*. Функция — это именованный фрагмент кода, отделенный от других. Она может принимать любое количество любых входных *параметров* и возвращать любое количество любых *результатов*.

С функцией можно сделать две вещи:

- *определить* ее, указав ноль или больше параметров;
- *вызвать* ее и получить ноль или больше результатов.

Определяем функцию с помощью ключевого слова def

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 символа подчеркивания и содержать только буквы, цифры или символ подчеркивания).

Будем 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 для демонстрации того, что функция ничего не делает. Оно эквивалентно утверждению «Эта страница специально оставлена пустой» (хотя теперь это и не так).

Вызываем функцию с помощью скобок

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 значение, которое она вернула, с помощью if:

```
>>> if agree():
...     print('Splendid!')
... else:
...     print('That was unexpected.')
...
Splendid!
```

Только что вы шагнули далеко вперед. Комбинация функций с проверками вроде if и циклами вроде while позволяет вам делать то, чего вы раньше не могли.

Аргументы и параметры

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.

Другими словами, значения вне функции называются *аргументами*, а значения внутри — *параметрами*.

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. А сейчас напишем функцию, которая принимает аргумент и что-то с ним делает. Адаптируем предыдущий фрагмент кода, который комментировал цвета. Назовем его `commentary` и сделаем так, чтобы он принимал в качестве аргумента строку `color`. Заставьте его вернуть описание строки вызывающей стороне, которая может решить, что с ней 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.

Затем вызывающая сторона присвоит строку переменной `comment`.

Что мы получили в результате?

```
>>> print(comment)
I've never heard of the color blue.
```

Функция может принимать любое количество аргументов (включая ноль) любого типа. Она может возвращать любое количество результатов (также включая ноль) любого типа. Если функция не вызывает `return` явно, вызывающая сторона получит результат `None`:

```
>>> print(do_nothing())
None
```

None — это полезно

`None` — это специальное значение в Python, которое заполняет собой пустое место, если функция ничего не возвращает. Оно не является булевым значением `False`, хоть и 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
```

Чтобы отличить `None` от булева значения `False`, используйте оператор `is`:

```
>>> thing = None
>>> if thing is None:
...     print("It's nothing")
... else:
...     print("It's something")
...
It's nothing
```

Разница кажется небольшой, однако в Python она важна: `None` потребуется вам, чтобы различать отсутствующие значения и пустые. Помните, что целочисленные нули, нули с плавающей точкой, пустые строки (`''`), списки (`[]`), кортежи (`(,)`), словари (`{}`) и множества (`set()`) все равны `False`, но это не то же самое, что `None`.

Напишем небольшую функцию, которая выводит на экран, является ли ее аргумент `None`, `True` или `False`:

```
>>> def whatis(thing):
...     if thing is None:
...         print(thing, "is None")
...     elif thing:
...         print(thing, "is True")
...     else:
...         print(thing, "is False")
...
```

Теперь выполним несколько проверок:

```
>>> whatis(None)
None is None
>>> whatis(True)
True is True
>>> whatis(False)
False is False
```

Как насчет реальных значений?

```
>>> whatis(0)
0 is False
>>> whatis(0.0)
0.0 is False
>>> whatis('')
 is False
>>> whatis("")
 is False
>>> whatis('''''')
 is False
>>> whatis(())
() is False
>>> whatis([])
[] is False
>>> whatis({})
{} is False
>>> whatis(set())
set() is False

>>> whatis(0.00001)
1e-05 is True
>>> whatis([0])
[0] is True
>>> whatis([''])
[''] is True
>>> whatis(' ')
  is True
```

Позиционные аргументы

Python довольно гибко обрабатывает аргументы функций в сравнении со многими другими языками. Наиболее распространенный тип аргументов — это *позиционные аргументы*, чьи значения копируются в соответствующие параметры по порядку.

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')
{'wine': 'chardonnay', 'entree': 'chicken', 'dessert': 'cake'}
```

Несмотря на распространенность аргументов такого типа, у них есть недостаток: 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'}
```

Можно 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 для параметров значения по умолчанию. Значения по умолчанию используются в том случае, если вызывающая сторона не предоставила соответствующий аргумент. Это стандартное действие оказывается достаточно полезным. Обратимся к предыдущему примеру:

```
>>> 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')
{'wine': 'chardonnay', 'entree': 'chicken', 'dessert': 'pudding'}
```

Если вы предоставите аргумент, он будет использован вместо аргумента по умолчанию:

```
>>> menu('dunkelfelder', 'duck', 'doughnut')
{'wine': 'dunkelfelder', 'entree': 'duck', 'dessert': 'doughnut'}
```

Значение параметров по умолчанию высчитывается, когда функция *определяется*, а не выполняется. Распространенной ошибкой новичков (а иногда и не совсем новичков) является использование изменяемого типа данных, таких как список или словарь, в качестве параметра по умолчанию.

В следующей проверке функция `buggy()` каждый раз должна 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 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-либо еще, указывающее 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']
```

Иногда этот вопрос задают на собеседованиях, связанных с Python. Теперь вы предупреждены.

Получаем/разбиваем аргументы — ключевые слова с помощью символа *

Если вы работали с языками программирования C или C++, то можете предположить, что астериск (`*`) в Python имеет какое-то отношение к *указателям*. Нет, у Python нет указателей.

Если символ `*` будет использован внутри функции с параметром, произвольное количество позиционных аргументов сгруппируется в один кортеж. В следующем примере `args` является кортежем параметров, который был создан из нуля или более 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 удобно при написании таких функций, как `print()`, которые принимают произвольное количество аргументов. Если в вашей функции имеются также *обязательные* позиционные аргументы, поместите их в начало — `*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 аргументов args, однако это распространенная идиома в Python. Также часто внутри функции используется конструкция *args, как это показано в предыдущем примере, несмотря на то что технически она является параметром и должна называться *params.

Подведем итоги.

- Можно передавать позиционный аргумент в функцию, которая соотнесет его с позиционным параметром. Подобные примеры вы уже видели в этой книге.
- Можно передать в функцию кортеж, и внутри он также останется кортежем. Это более простой вариант предыдущей ситуации.
- Можно передать позиционные аргументы в функцию и объединить их в параметр `*args`, который будет развернут в кортеж `args`. Эту ситуацию мы описали в текущем подразделе.
- Можно «разбить» кортеж с именем `args` на позиционные параметры `*args` внутри функции, которые затем будут пересобраны в кортеж-параметр `args`:

```
>>> print_args(2, 5, 7, 'x')
Positional tuple: (2, 5, 7, 'x')
>>> args = (2,5,7,'x')
>>> print_args(args)
```

```
Positional tuple: ((2, 5, 7, 'x'),)
>>> print_args(*args)
Positional tuple: (2, 5, 7, 'x')
```

Символ * можно использовать только при описании функции и ее вызове:

```
>>> *args
  File "<stdin>", line 1
SyntaxError: can't use starred expression here
```

Итак:

- находясь за пределами функции, `*args` разбивает кортеж `args` на разделенные запятыми позиционные параметры;
- будучи внутри функции, `*args` собирает все позиционные аргументы в единый кортеж `args`. Вы можете использовать имена `*params` и `params`, но чаще всего имя `*args` используется как для аргументов, так и для параметров.

Читатели, обладающие синестезией, могут неявно услышать, что `*args` произносится как *выдох-`args`* снаружи и *вдох-`args`* изнутри.

Получаем/разбиваем аргументы — ключевые слова с помощью символов **

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()
Keyword arguments: {}
>>> print_kwargs(wine='merlot', entree='mutton', dessert='macaroon')
Keyword arguments: {'dessert': 'macaroon', 'wine': 'merlot', 'entree': 'mutton'}
```

Внутри функции `kwargs` является параметром-словарем.

Порядок аргументов будет следующим:

- обязательные позиционные аргументы;
- необязательные позиционные аргументы (`*args`);
- необязательные аргументы — ключевые слова (`**kwargs`).

Как и в случае с `args`, вам не обязательно называть этот аргумент `kwargs`, однако такова распространенная практика[1].

[1] Имена Args и Kwargs очень хорошо подошли бы пиратским попугаям.

Синтаксис ** можно применять только при вызове функции или ее определении[1]:

```
>>> **kwparams
  File "<stdin>", line 1
    **kwparams
     ^
SyntaxError: invalid syntax
```

Подведем итоги:

- вы можете передавать аргументы в функцию по ключевым словам — они будут соотнесены с параметрами. Вы уже видели этот подход на практике;
- вы можете передать в функцию словарь — внутри он также будет представлять собой словарь с параметрами. Это частный случай предыдущей ситуации;
- вы можете передать в функцию один или несколько аргументов по ключевым словам (в формате *имя=значение*) и собрать их внутри в виде конструкции **kwargs, которая будет преобразована в параметр-словарь с именем kwargs. Такую ситуацию мы рассмотрели в этом подразделе;
- находясь за пределами функции, **kwargs *разбивает* словарь kwargs на аргументы в формате *имя=значение*;
- находясь внутри функции, **kwargs *собирает* аргументы в формате *имя=значение* в словарь с именем kwargs.

Если вам помогает аналогия со слуховыми образами, представьте, что при разбиении аргументов для каждой звездочки делается выдох, а при их сборе — вдох.

Аргументы, передаваемые только по ключевым словам

В функцию можно передать аргумент — ключевое слово с тем же самым именем, что и у позиционного параметра. Однако, скорее всего, это приведет не к тому результату, который вы ожидаете. Python 3 позволяет указывать аргументы, которые передаются *только по ключевым словам*. Соответственно названию их нужно предоставлять в формате *имя=значение*, а не позиционно, как значение. Знак * в определении функции говорит о том, что параметры start и end нужно предоставлять как именованные аргументы, если мы не хотим использовать их значения по умолчанию:

```
>>> def print_data(data, *, start=0, end=100):
...     for value in (data[start:end]):
...         print(value)
...
>>> data = ['a', 'b', 'c', 'd', 'e', 'f']
>>> print_data(data)
a
```

[1] А также, начиная с версии Python 3.5, при объединении словарей с помощью конструкции {**a, **b}, которую вы видели в главе 8.

```
b
c
d
e
f
>>> print_data(data, start=4)
e
f
>>> print_data(data, end=2)
a
b
```

Изменяемые и неизменяемые аргументы

Вы не забыли, что, если присвоить один список двум переменным, вы сможете изменить его, используя любую из этих переменных? И что это не сработает, если обе переменные ссылаются, например, на целое число или строку? Это происходит потому, что список является изменяемым, а строка и целое число — нет. Вам нужно помнить про эту особенность при передаче аргументов в функцию. Если аргумент изменяемый, то его значение можно изменить *изнутри функции* с помощью соответствующего параметра[1]:

```
>>> outside = ['one', 'fine', 'day']
>>> def mangle(arg):
...     arg[1] = 'terrible!'
...
>>> outside
['one', 'fine', 'day']
>>> mangle(outside)
>>> outside
['one', 'terrible!', 'day']
```

Хорошим тоном считается умение избегать подобных ситуаций[2]. Любой документ, являющийся аргументом, может измениться или вернуть новое значение.

Строки документации

В Python *удобочитаемость имеет значение*. Вы можете прикрепить документацию к определению функции, включив в начало ее тела строку, которая и называется *строкой документации*:

```
>>> def echo(anything):
...     'echo returns its input argument'
...     return anything
```

---

1 Как в подростковых фильмах ужасов, когда герои узнают: «Звонок идет изнутри дома!»

2 Как в старом врачебном анекдоте: «Мне больно, когда я делаю вот так». — «Значит, больше так не делайте».

Вы можете сделать строку документации довольно длинной и даже применить к ней, если хотите, форматирование. Это 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 вывода строки документации функции вызовите 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 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. Двойное нижнее подчеркивание (на сленге его называют *dunder*) часто используется для именования внутренних переменных Python, поскольку программисты, скорее всего, не будут использовать подобные имена переменных.

Функции — это объекты первого класса

Я уже упоминал мантру Python: *все является объектом*. Числа, строки, кортежи, списки, словари и даже функции — это объекты. Функции в Python выступают объектами первого класса. Вы можете присвоить их переменным, использовать как аргументы для других функций и возвращать из функций. Это дает возможность решать в Python такие задачи, которые трудно выполнить во многих других языках.

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 с именем `func`, представляющий собой функцию, которую нужно запустить и которая просто вызывает другую функцию:

```
>>> def run_something(func):
...     func()
```

Если мы передадим `answer` в функцию `run_something()`, то используем ее в качестве данных, 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 функция является объектом, как и все остальное в Python:

```
>>> 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. Она встроена в Python и высчитывает сумму значений итерабельного числового (целочисленного или с плавающей точкой) аргумента.

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 их можно применять даже в качестве ключей 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 в свой аргумент:

```
>>> 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. Обе они могут изменяться и запоминать значения переменных, которые были созданы вне функции.

Обратимся еще раз к примеру `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). Получается своего рода 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 создавались функцией `knights2`:

```
>>> a()
"We are the knights who say: 'Duck'"
>>> b()
"We are the knights who say: 'Hasenpfeffer'"
```

Анонимные функции: лямбда-выражения

В Python *лямбда-функция* — это анонимная функция, выраженная в виде одного оператора. Ее можно использовать вместо обычной небольшой функции.

Чтобы проиллюстрировать анонимные функции, сначала создадим пример, в котором используются обычные функции. В первую очередь определим функцию `edit_story()`. Она имеет следующие аргументы:

- `words` — список слов;
- `func` — функцию, 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 запишет каждое слово с большой буквы 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-выражение имеет ноль или больше аргументов, разделенных запятой, за которыми следуют двоеточие (`:`) и определение функции. Этому лямбда-выражению мы передаем один аргумент `word`. Для вызова лямбда-выражения не используются круглые скобки, в отличие от вызова функции, созданной с помощью ключевого слова `def`.

Зачастую использование реальных функций, таких как `enliven()`, гораздо понятнее, 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 можно найти в главе 20.

Генераторы

В Python *генератор* — это объект, который предназначен для создания последовательностей. С его помощью вы можете итерировать потенциально огромные последовательности, не создавая и не сохраняя всю последовательность в памяти сразу. Генераторы часто становятся источником данных для итераторов. Как вы помните, один из них, `range()`, мы уже использовали 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 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 в том же состоянии.

Функции-генераторы

Если вы хотите создать потенциально большую последовательность, вы можете написать *функцию-генератор*. Это обычная функция, однако 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 обычная 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
```

Генератор можно запустить лишь однажды. Списки, множества, строки и словари существуют в памяти, а генераторы создают свои значения на лету и выдают их по одному с помощью итератора. Генератор не запоминает значения, поэтому вы не можете перезапустить его или создать резервную копию.

Если вы попробуете снова проитерировать по генератору, вы увидите, что он истощился:

```
>>> for try_again in ranger:
...     print(try_again)
...
>>>
```

Включения генераторов

Вы уже сталкивались с включениями списков, словарей и множеств. *Включение генератора* выглядит так же, но заключается в круглые скобки, а не в квадратные или фигурные. Оно похоже на сокращенную функцию-генератор, которая неявно выполняет `yield` и возвращает объект генератора:

```
>>> genobj = (pair for pair in zip(['a', 'b'], ['1', '2']))
>>> genobj
<generator object <genexpr> at 0x10308fde0>
>>> for thing in genobj:
...     print(thing)
...
('a', '1')
('b', '2')
```

Декораторы

Иногда вам нужно модифицировать существующую функцию, не меняя при этом ее исходный код. Чаще всего добавляется выражение для отладки, чтобы посмотреть, какие аргументы были переданы.

Декоратор — это функция, которая принимает одну функцию в качестве аргумента и возвращает другую функцию. Давайте рассмотрим, какие приемы из нашего арсенала мы можем использовать:

- `*args` и `**kwargs`;
- внутренние функции;
- функции в качестве аргументов.

Функция `document_it()` определяет декоратор, который:

- выведет имя функции и значения 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 с дополнительными выражениями, которые добавляет `document_it()`. Декоратор не обязательно должен запускать код функции `func`, но функция `document_it()` вызовет часть `func`, чтобы вы получили 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)  # создание декоратора вручную
>>> cooler_add_ints(3, 5)
Running function: add_ints
Positional arguments: (3, 5)
Keyword arguments: {}
Result: 8
8
```

В качестве альтернативы созданию декоратора вручную (этот процесс мы только что рассмотрели) можно добавить конструкцию *`@имя_декоратора`* перед функцией, которую нужно декорировать:

```
>>> @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
8
```

У каждой функции может быть больше одного декоратора. Напишем еще один с именем `square_it()`, который 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 выполнится тот, что находится сразу над ним. Любой порядок вызова вернет один и тот же конечный результат, но вы можете увидеть все промежуточные этапы:

```
>>> @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 изменить 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

С тем, кто в искусстве больше преуспел...

Уильям Шекспир

Имя может ссылаться на несколько разных объектов в зависимости от того, где оно используется. Программы на Python имеют различные *пространства имен* — разделы, внутри которых определенное имя уникально и не связано с таким же именем в других пространствах имен.

Каждая функция определяет собственное пространство имен. Если вы определите переменную с именем `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 в других пространствах имен разными способами.

В основной программе определяется *глобальное* пространство имен, поэтому переменные, находящиеся в нем, являются *глобальными*.

Значение глобальной переменной можно получить 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 вы попробуете получить значение глобальной переменной *и* изменить его внутри функции, то 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_print_global
UnboundLocalError: local variable 'animal' referenced before assignment
```

Если вы просто измените его, то 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 же произошло? Сначала мы присвоили строку `'fruitbat'` глобальной переменной с именем `animal`. Функция `change_local()` также имеет переменную с именем `animal`, но та находится в ее локальном пространстве имен.

Я использовал 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 не к локальной переменной внутри функции, а к глобальной переменой, нужно явно использовать ключевое слово `global` (вы знали, что я это скажу: *явное лучше, чем неявное*):

```
>>> 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, таким образом, 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 как 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 символов _ и __ в именах

Имена, которые начинаются с двух нижних подчеркиваний (`__`) и заканчиваются ими, зарезервированы для использования внутри Python, поэтому вам не следует применять их к своим собственным переменным. Подобный шаблон именования был выбран потому, что разработчики, скорее всего, не стали бы им пользоваться для создания имен своих переменных.

Например, имя функции находится в системной переменной `function.__name__`, а имя ее строки документации — в `function.__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__`.

Рекурсия

До сих пор мы видели функции, которые выполняют какие-то определенные действия и, возможно, вызывают другие функции. Но что, если функция вызовет саму себя?[1] Это называется *рекурсией*. Так же как не нужны бесконечные циклы `while`

[1] Это как сказать: «Я хочу, чтоб мне давали монетку каждый раз, когда я хочу, чтобы мне дали монетку».

или `for`, не нужна и бесконечная рекурсия. Стоит ли нам волноваться за целостность пространственно-временного континуума?

Python снова спасает Вселенную, генерируя исключение, если вы зайдете слишком далеко:

```
>>> def dive():
...     return dive()
...
>>> dive()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in dive
  File "<stdin>", line 2, in dive
  File "<stdin>", line 2, in dive
  [Previous line repeated 996 more times]
RecursionError: maximum recursion depth exceeded
```

Рекурсия может быть полезна, если вы работаете с чем-то вроде списков, содержащих списки, которые содержат списки.

Предположим, вы хотите «уплотнить» все подсписки[1] независимо от того, насколько глубоко они вложены. Для этого отлично подойдет функция-генератор:

```
>>> def flatten(lol):
...     for item in lol:
...         if isinstance(item, list):
...             for subitem in flatten(item):
...                 yield subitem
...         else:
...             yield item
...
>>> lol = [1, 2, [3,4,5], [6,[7,8,9], []]]
>>> flatten(lol)
<generator object flatten at 0x10509a750>
>>> list(flatten(lol))
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

В Python 3.3 появилось выражение `yield from`, которое позволяет генератору передавать часть работы другому генератору. Мы можем использовать его для того, чтобы упростить функцию `flatten()`:

```
>>> def flatten(lol):
...     for item in lol:
...         if isinstance(item, list):
...             yield from flatten(item)
...         else:
...             yield item
...
>>> lol = [1, 2, [3,4,5], [6,[7,8,9], []]]
>>> list(flatten(lol))
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

[1] Это еще один популярный вопрос на собеседованиях. Давайте соберем всю коллекцию!

Асинхронные функции

Ключевые слова `async` и `await` были добавлены в Python 3.5 для того, чтобы можно было определять и запускать *асинхронные функции*. Их можно описать так:

- они относительно новые;
- они значительно отличаются друг от друга, чтобы их сложнее было понять;
- они станут более важными и более изученными со временем.

По этим причинам я перенес рассмотрение вопросов, связанных с асинхронностью, в приложение B. Сейчас вам нужно знать, что, если вы в объявлении функции перед словом `def` видите слово `async`, это значит, что функция асинхронная. Аналогично, если перед вызовом функции вы видите слово `await`, эта функция также является асинхронной. Основное различие между асинхронными и обычными функциями заключается в том, что асинхронные могут «передавать управление» вместо того, чтобы заставлять основной поток выполнения ждать до конца их выполнения.

Исключения

В некоторых языках программирования ошибки отображаются с помощью специальных возвращаемых значений. В Python, если дело пахнет жареным[1], используется *исключение*: код, который выполняется, когда происходит связанная с ним ошибка.

Какие-то из исключений вы уже видели, пытаясь получить доступ к позиции, не входящей в список/кортеж, или к словарю с несуществующим ключом. Когда вы выполняете код, который при определенных условиях может не сработать, вам также понадобятся *обработчики исключений* для перехвата любых потенциальных ошибок.

Чтобы пользователь знал о происходящем, рекомендуется добавлять обработчики исключений везде, где исключение может быть сгенерировано. Не всегда у вас получится исправить ошибку, но вы хотя бы будете знать, при каких обстоятельствах она появилась, и аккуратно завершите программу. Если исключение сгенерировалось в функции и там не обработалось, оно будет *всплывать* до тех пор, пока его не поймает соответствующий обработчик в одной из вызывающих функций. Если вы не предоставите собственный обработчик исключений, 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
```

[1] Интересно, это выражение характерно только для северного полушария? Говорят ли австралийцы и новозеландцы«дело пахнет вареным», когда все идет не так, как хотелось бы?

Обрабатываем ошибки с помощью операторов try и except

Делай или не делай. Не надо пытаться.

Йода

Вместо того чтобы полагаться на волю случая, размещайте свой код в блоке `try` и используйте блок `except` для обработки ошибки:

```
>>> 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` пропускается.

Отсутствие аргументов в блоке `except`, как в предыдущем примере, позволяет ловить исключения любого типа. Если предполагается, что могут возникнуть несколько типов исключений, лучшим решением будет предоставить отдельный обработчик для каждого из них. Никто не заставляет это делать — можно использовать блок `except` без аргументов, но тогда обработка будет достаточно поверхностной (что-то вроде вывода на экран строки Произошла ошибка). Разрешается использовать любое количество обработчиков исключений.

Иногда вам будут нужны и другие подробности исключения, а не только его тип. Вы можете получить объект исключения целиком в переменной `имя`, если используете такую 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`, для демонстрации того, 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, универсальном коде `except`.

Создаем собственные исключения

В предыдущем подразделе обсуждалась обработка исключений, но все исключения (такие как `IndexError`) заранее определены в Python или его стандартных библиотеках. Вы можете использовать любые из этих исключений, но можете также и определять собственные типы исключений, чтобы обрабатывать особые ситуации, которые могут возникнуть в ваших программах.

Для этого потребуется определить новый тип объекта с помощью *класса*. Заниматься классами мы начнем в главе 10, поэтому, если вы с ними не знакомы, можете обратиться к соответствующему разделу.

Любое исключение является классом, частным случаем класса `Exception`. Создадим исключение, которое называется `UppercaseException` и которое будет вызываться при нахождении слова, записанного буквами 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 какое-то поведение для `UppercaseException` (обратите внимание — мы просто использовали `pass`), позволив его родительскому классу `Exception` самостоятельно разобраться, что именно выводить 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
```

Читайте далее

Объекты! В книге, посвященной объектно-ориентированному языку, мы когда-нибудь обязаны были до них добраться.

Упражнения

9.1. Определите функцию `good()`, которая возвращает следующий список: `['Harry', 'Ron', 'Hermione']`.

9.2. Определите функцию генератора `get_odds()`, которая возвращает нечетные числа из диапазона `range(10)`. Используйте цикл `for`, чтобы найти и вывести третье возвращенное значение.

9.3. Определите декоратор `test`, который выводит строку `'start'` при вызове функции и строку `'end'`, когда функция завершает свою работу.

9.4. Определите исключение `OopsException`. Сгенерируйте его и посмотрите, что произойдет. Затем напишите код, позволяющий поймать это исключение и вывести строку `'Caught an oops'`.

ГЛАВА 10
Ой-ой-ой: объекты и классы

> Таинственных объектов не бывает. Они такими просто кажутся.
>
> *Элизабет Боуэн*

> Возьмите объект. Сделайте с ним что-нибудь. Добавьте к нему что-нибудь еще.
>
> *Джаспер Джонс*

Как я уже говорил ранее, в Python все, от чисел до функций, является объектами. Однако Python скрывает бóльшую часть функционирования объектных механизмов с помощью особого синтаксиса. Вы можете написать `num = 7`, и, таким образом, у вас появится объект типа `int` со значением `7` и назначится ссылка на объект по имени `num`. Заглядывать внутрь объектов нужно только в случае, если вам необходимо создать собственный объект или модифицировать поведение уже существующих. В этой главе вы увидите, как сделать и то и другое.

Что такое объекты

Объект — это пользовательская структура данных, которая содержит как данные (переменные, называемые *атрибутами*), так и код (функции, называемые *методами*). Он представляет собой уникальный экземпляр чего-то конкретного. Можно думать об объектах как о существительных, а об их методах — как о глаголах. Объект представляет собой единичный экземпляр, методы определяют его взаимодействие с другими объектами.

Например, целочисленный объект со значением `7` может использовать такие методы, как сложение и умножение, что показано в главе 3. `8` — другой объект. Это значит, что существует класс `Integer`, встроенный в Python, которому принадлежат объекты `7` и `8`. Строки `'cat'` и `'duck'` также являются объектами и имеют

методы, с которыми вы уже познакомились в главе 5, — например, `capitalize()` и `replace()`.

В отличие от модулей у вас одновременно может быть несколько объектов (также называемых *экземплярами*), каждый из которых может иметь самые разные атрибуты. Объекты — как суперструктуры данных, в которые был добавлен код.

Простые объекты

Начнем мы с простых классов, а разговор о наследовании перенесем на несколько страниц вперед.

Определяем класс с помощью ключевого слова class

Чтобы создать новый объект, который никто раньше не создавал, вам сначала нужно определить класс, который укажет на то, что в объекте содержится.

В главе 2 я сравнил объект с пластмассовой коробкой. *Класс* похож на форму, по образцу которой создается эта коробка. Например, в Python есть встроенный класс, создающий строковые объекты, такие как `'cat'` и `'duck'`. Python имеет и множество других встроенных классов, позволяющих создавать другие стандартные типы данных, в том числе списки, словари и т. д. Чтобы создать собственный объект в Python, вам сначала нужно определить класс с помощью ключевого слова `class`. Рассмотрим несколько простых примеров.

Предположим, вы хотите определить объекты для представления информации о котах[1]. Каждый объект будет представлять одного кота. Сначала вам нужно определить класс `Cat` в качестве формы. В последующих примерах мы попробуем использовать больше версий этого класса, продвигаясь от простейшего класса к тому, который действительно может делать что-то полезное.

Мы следуем концепциям об именовании, указанным в PEP-8 (https://oreil.ly/gAJOF).

Для начала создадим самый простой из возможных классов — пустой:

```
>>> class Cat():
...     pass
```

Можно и так:

```
>>> class Cat:
...     pass
```

Как и в случае с функциями, нам нужно сказать `pass`, чтобы указать, что этот класс пуст. Такое определение является минимально необходимым для создания

[1] Мы это сделаем, даже если вы не хотите.

объекта. Вы создаете объект из класса, вызывая имя класса так, как если бы это была функция:

```
>>> a_cat = Cat()
>>> another_cat = Cat()
```

В этом случае вызов `Cat()` создаст два отдельных объекта класса `Cat`, и мы присвоим им имена `a_cat` и `another_cat`. Но наш класс `Cat` пуст, поэтому объекты, которые мы создали, просто занимают место и ничего не делают.

Хотя нет, кое-что они могут.

Атрибуты

Атрибут — это переменная, расположенная внутри класса или объекта. Во время и после создания объекта или класса вы можете присвоить им атрибуты. Атрибутом может быть любой другой объект. Давайте снова создадим два объекта типа `Cat`:

```
>>> class Cat:
...     pass
...
>>> a_cat = Cat()
>>> a_cat
<__main__.Cat object at 0x100cd1da0>
>>> another_cat = Cat()
>>> another_cat
<__main__.Cat object at 0x100cd1e48>
```

Когда мы определили класс `Cat`, мы не указали, в каком виде его объект будет выведен на экран. Тут в дело вмешивается Python и выводит на экран что-то вроде `<__main__.Cat object at 0x100cd1da0>`. В разделе «Магические методы» на с. 215 вы увидите, как можно изменить это поведение, используемое по умолчанию.

Теперь назначим несколько атрибутов нашему первому объекту:

```
>>> a_cat.age = 3
>>> a_cat.name = "Mr. Fuzzybuttons"
>>> a_cat.nemesis = another_cat
```

Можем ли мы получить доступ к этим значениям? Надеюсь, что да:

```
>>> a_cat.age
3
>>> a_cat.name
'Mr. Fuzzybuttons'
>>> a_cat.nemesis
<__main__.Cat object at 0x100cd1e48>
```

Поскольку `nemesis` был атрибутом, ссылающимся на другой объект `Cat`, мы можем использовать атрибут `a_cat.nemesis` для доступа к нему. Но у этого другого объекта еще нет атрибута `name`:

```
>>> a_cat.nemesis.name
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Cat' object has no attribute 'name'
```

Дадим имя нашему коту:

```
>>> a_cat.nemesis.name = "Mr. Bigglesworth"
>>> a_cat.nemesis.name
'Mr. Bigglesworth'
```

Даже в подобных простейших объектах можно хранить несколько атрибутов. Так что для хранения разных значений допускается использование нескольких объектов вместо чего-то наподобие списка или словаря.

Когда вы слышите слово *«атрибуты»*, речь, скорее всего, идет об атрибутах объектов. Помимо этого, существуют также *атрибуты класса*. Чем они различаются, вы увидите в подразделе «Атрибуты классов и объектов» на с. 210.

Методы

Метод — это функция класса или объекта. Метод выглядит так же, как и любая другая функция, но может быть использован особым образом — этот вопрос мы рассмотрим далее в этой главе, в подразделе «Свойства для доступа к атрибутам» и в разделе «Типы методов».

Инициализация

Если вы хотите присвоить атрибуты объекту во время его создания, вам понадобится специальный метод инициализации `__init__`:

```
>>> class Cat:
...     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 с именем `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 `Cat` создает объект, который ничего не делает. Наша третья попытка покажет, насколько легко создать простой объект в Python и присвоить значение одному из его атрибутов. В этот раз мы добавим параметр `name` в метод инициализации:

```
>>> class Cat():
...     def __init__(self, name):
...         self.name = name
...
>>>
```

[1] Вы встретите множество примеров двойных подчеркиваний в именах Python. Для экономии времени некоторые люди называют их dunder.

Теперь мы можем создать объект класса `Cat`, передав строку для параметра `name`:

```
>>> furball = Cat('Grumpy')
```

Эта строка кода делает следующее:

- выполняет поиск определения класса `Cat`;
- *инстанцирует* (создает) новый объект в памяти;
- вызывает метод объекта `__init__()`, передавая только что созданный объект под именем `self` и другой аргумент (`'Grumpy'`) в качестве значения параметра `name`;
- сохраняет в объекте значение переменной `name`;
- возвращает новый объект;
- прикрепляет к объекту переменную `furball`.

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 в качестве результата.

А что со значением `name`, которое мы передали? Оно было сохранено как атрибут объекта. Вы можете прочитать и записать его непосредственно:

```
>>> print('Our latest addition: ', furball.name)
Our latest addition: Grumpy
```

Помните, *внутри* определения класса `Cat` вы получаете доступ к атрибуту `name` с помощью конструкции `self.name`. Когда вы создаете реальный объект и присваиваете его переменной вроде `furball`, то ссылаетесь на этот атрибут как `furball.name`.

Не обязательно иметь метод `__init__()` в описании каждого класса — он используется для того, чтобы различать объекты одного класса. Это не то, что в некоторых других языках называется конструктором. Python уже создал объект для вас. Метод `__init__()` следует рассматривать скорее как *средство инициализации*.

Вы можете создать много отдельных объектов из одного класса. Но помните, что в Python данные реализованы как объекты, поэтому и сам класс является объектом. Однако в вашей программе может существовать только один объект для каждого класса. Если класс Cat уже определен, как мы сделали в этой главе, то второй такой класс создать не получится. Он становится чем-то вроде Горца — должен остаться только один.

Наследование

Предположим, вы пытаетесь решить какую-то задачу и узнаете, что уже существует класс, создающий объекты, которые делают почти все из того, что вам нужно. Как поступить? Вы можете модифицировать данный класс: однако при этом вы сделаете его более сложным и, вполне вероятно, испортите то, что раньше работало.

Или 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, а части старого и нового классов, которые раньше работали одинаково, могут оказаться непохожими друг на друга, поскольку теперь находятся в разных местах.

Одним из решений проблемы является *наследование* — создание нового класса *из* уже существующего, но с некоторыми дополнениями и изменениями. Это хороший способ использовать код повторно. Когда вы применяете наследование, новый класс может автоматически использовать весь код старого класса, но вам при этом не нужно его копировать.

Наследование от родительского класса

Вы определяете только то, что вам нужно добавить или изменить в новом классе, и этот код переопределяет поведение старого класса. Оригинальный класс называется *родительским классом*, *суперклассом* или *базовым классом*, а новый класс именуется *потомком*, *подклассом* или *производным классом*. Эти термины в объектно-ориентированном программировании взаимозаменяемы.

Итак, попробуем что-нибудь унаследовать. В следующем примере мы определим пустой класс, который называется `Car`. Далее мы определим подкласс класса `Car`, который называется `Yugo`[1]. Вы определяете подкласс с помощью все того же ключевого слова `class`, но указываете внутри скобок имя родительского класса `class Yugo(Car)`, как показано ниже:

```
>>> class Car():
...     pass
...
>>> class Yugo(Car):
...     pass
...
```

Проверить, унаследован ли класс от другого класса, можно с помощью функции `issubclass()`:

```
>>> issubclass(Yugo, Car)
True
```

Далее создадим объекты каждого класса:

```
>>> give_me_a_car = Car()
>>> give_me_a_yugo = Yugo()
```

Производный класс выступает уточненной версией родительского класса: если говорить в терминах объектно-ориентированных языков, `Yugo` является `Car`. Объект с именем `give_me_a_yugo` — это экземпляр класса `Yugo`, но он также наследует все то, что может делать класс `Car`. В нашем случае классы `Car` и `Yugo` абсолютно

[1] Недорогая и не очень качественная машина 1980-х годов.

бесполезны, поэтому попробуем 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`, а это может привести к кризису самоопределения. Посмотрим, как решить проблему.

У такого приема, как наследование, есть определенные преимущества, но иногда им пользуются слишком часто. Многолетний опыт работы в объектно-ориентированном программировании показал, что чрезмерное увлечение наследованием может затруднить управление программами. Вместо этого рекомендуется сделать акцент на использовании других приемов, таких как агрегирование и композиция. Их мы рассмотрим также в этой главе.

Переопределение методов

Как вы только что увидели, новый класс изначально наследует все, что находится в его родительском классе. Далее вы узнаете, как можно заменить, или переопределить, 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 переопределили метод `exclaim()`. Переопределять можно любые методы, включая `__init__()`. Вот еще один пример — с классом `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 по-разному сохраняет значение переменной `name` внутри объекта:

```
>>> 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 производный класс можно *добавить* и метод, которого не было в родительском классе. Вернемся к классам `Car` и `Yugo` и 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 то, чего не может делать класс `Car`, и, таким образом, мы ясно видим отличительные особенности класса `Yugo`.

Получаем помощь от своего родителя с использованием метода super()

Мы видели, как производный класс может добавить или переопределить метод родительского класса. Но что, если нужно будет вызвать этот родительский метод? «Рад, что вы спросили», — говорит метод `super()`. Сейчас мы определим новый класс `EmailPerson`, который представляет объект класса `Person` с адресом 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 просто дать ему любые необязательные 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 это сделать, но тогда потеряли бы возможность применять наследование. Мы использовали метод `super()`, чтобы заставить `Person` делать свою работу так же, как это делает обычный 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 по-своему, но все еще нуждается в помощи родителя (как и в реальной жизни).

Множественное наследование

Вы только что видели несколько примеров классов, у которых нет родительского класса, и несколько примеров, у которых есть. Однако объекты могут наследовать и от нескольких родительских классов.

Если ваш класс ссылается на метод или атрибут, которого у него нет, Python будет искать его в родительском классе. Что, если у нескольких классов будут атрибуты с одинаковыми именами? Кто победит?

В отличие от наследования у людей, когда доминантный ген побеждает независимо от того, кому он принадлежит, наследование в Python зависит от *порядка разрешения методов*. Каждый класс Python имеет особый метод `mro()`, возвращающий список классов, в которых будет выполнен поиск метода или атрибута для объекта этого класса. Похожий атрибут с именем `__mro__` представляет собой кортеж, содержащий имена этих классов. Как и в плей-офф, когда действует принцип «внезапной

смерти», побеждает первый. Сейчас мы определим родительский класс `Animal`, два класса-потомка (`Horse` и `Donkey`), а затем еще два класса, полученных из них[1]:

```
>>> class Animal:
...     def says(self):
            return 'I speak!'
...
>>> class Horse(Animal):
...     def says(self):
...         return 'Neigh!'
...
>>> class Donkey(Animal):
...     def says(self):
...         return 'Hee-haw!'
...
>>> class Mule(Donkey, Horse):
...     pass
...
>>> class Hinny(Horse, Donkey):
...     pass
...
```

Если нам понадобится метод или атрибут класса `Mule`, Python будет искать его в следующих классах по порядку.

1. Сам объект (типа `Mule`).
2. Класс объекта (`Mule`).
3. Первый родительский класс этого класса (`Donkey`).
4. Второй родительский класс этого класса (`Horse`).
5. Прародительский класс этого класса (`Animal`).

Для класса `Hinny` список выглядит примерно так же, только класс `Horse` расположится перед классом `Donkey`:

```
>>> Mule.mro()
[<class '__main__.Mule'>, <class '__main__.Donkey'>,
<class '__main__.Horse'>, <class '__main__.Animal'>,
<class 'object'>]
>>> Hinny.mro()
[<class '__main__.Hinny'>, <class '__main__.Horse'>,
<class '__main__.Donkey'>, <class '__main__.Animal'>,
class 'object'>]
```

Так что же говорят эти прекрасные животные?

```
>>> mule = Mule()
>>> hinny = Hinny()
>>> mule.says()
'hee-haw'
>>> hinny.says()
'neigh'
```

[1] У мула (mule) папа — осел, а мама — лошадь. У лошака (hinny) папа — конь, а мама — ослица.

Мы перечислили родительские классы в порядке «папа, мама», чтобы они разговаривали как «папы». Если бы у классов `Horse` и `Donkey` не было метода `says()`, классы `Mule` и `Hinny` использовали бы метод `says()` прародительского класса `Animal` и возвратили бы значение `'I speak!'`.

Примеси

Вы можете включить дополнительный родительский класс в определение вашего класса, но только в качестве вспомогательного. Таким образом, он не будет иметь общих методов с другими родительскими классами, что позволит избежать неоднозначности при разрешении методов, о которой я говорил ранее.

Такой родительский класс называют классом-*примесью* или *миксином*. Примеси можно использовать для выполнения «сторонних» задач, таких, например, как ведение журнала. Рассмотрим класс-примесь, который форматирует и выводит на экран атрибуты объекта:

```
>>> class PrettyMixin():
...     def dump(self):
...         import pprint
...         pprint.pprint(vars(self))
...
>>> class Thing(PrettyMixin):
...     pass
...
>>> t = Thing()
>>> t.name = "Nyarlathotep"
>>> t.feature = "ichor"
>>> t.age = "eldritch"
>>> t.dump()
{'age': 'eldritch', 'feature': 'ichor', 'name': 'Nyarlathotep'}
```

В защиту self

Помимо применения пробелов, Python критикуют и за то, что необходимо включать `self` в качестве первого аргумента методов экземпляра класса (методов, которые вы видели в предыдущих примерах). Python использует аргумент `self`, чтобы найти атрибуты и методы правильного объекта. Например, я покажу, как вы можете вызвать метод объекта и что Python при этом сделает за кулисами на самом деле.

Помните класс `Car` из предыдущих примеров? Снова вызовем метод `exclaim()`:

```
>>> a_car = Car()
>>> a_car.exclaim()
I'm a Car!
```

Вот что происходит за кулисами Python.

- Выполняется поиск класса (`Car`) объекта `a_car`.
- Объект `a_car` передается методу `exclaim()` класса `Car` как параметр `self`.

Ради развлечения вы и сами можете запустить пример таким образом, и он сработает точно так же, как и нормальный синтаксис (`a_car.exclaim()`):

```
>>> Car.exclaim(car)
I'm a Car!
```

Тем не менее нет никаких причин использовать этот более длинный стиль.

Доступ к атрибутам

В Python атрибуты объектов и методы обычно являются общедоступными, поэтому предполагается, что тот, кто их использует, не станет этим злоупотреблять (это иногда называют «установкой на взрослых людей»). Давайте сравним прямой доступ и его альтернативы.

Прямой доступ

Как вы уже видели, вы можете получать и задавать значения атрибутов напрямую:

```
>>> class Duck:
...     def __init__(self, input_name):
...         self.name = input_name
...
>>> fowl = Duck('Daffy')
>>> fowl.name
'Daffy'
```

Но что случится, если кто-то решит этим злоупотребить?

```
>>> fowl.name = 'Daphne'
>>> fowl.name
'Daphne'
```

В следующих двух подразделах показаны способы обеспечения определенной конфиденциальности атрибутов, значения которых не должны быть случайно изменены.

Геттеры и сеттеры

Отдельные объектно-ориентированные языки поддерживают закрытые атрибуты объектов, к которым нельзя получить доступ напрямую. Программистам часто приходится писать *геттеры* и *сеттеры*, чтобы прочитать и записать значения таких атрибутов.

В Python нет закрытых атрибутов, но вы можете написать геттеры и сеттеры для атрибутов с обфусцированными именами, чтобы создать нечто подобное (Лучшим решением будет использование *свойств*, они описаны в следующем подразделе.)

Далее в примере мы определим класс `Duck`, имеющий один атрибут экземпляра `hidden_name`. Мы не хотим, чтобы кто-то мог обратиться к атрибуту напрямую, по-

этому определим два метода: геттер (`get_name()`) и сеттер (`set_name()`). К каждому из них обращается свойство с именем `name`. Я добавил выражение `print()` в оба метода, чтобы показать момент его вызова:

```
>>> 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

>>> don = Duck('Donald')
>>> don.get_name()
inside the getter
'Donald'
>>> don.set_name('Donna')
inside the setter
>>> don.get_name()
inside the getter
'Donna'
```

Свойства для доступа к атрибутам

Более питонским решением для закрытия атрибутов является использование *свойств*.

Существует два способа сделать это. Первый — добавить конструкцию `name = property(get_name, set_name)` последней строкой в предыдущее определение класса `Duck`:

```
>>> 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)
```

Старый подход с геттерами и сеттерами тоже сработает:

```
>>> don = Duck('Donald')
>>> don.get_name()
inside the getter
'Donald'
>>> don.set_name('Donna')
inside the setter
```

```
>>> don.get_name()
inside the getter
'Donna'
```

Но теперь вы также можете использовать свойство `name`, чтобы получить и установить скрытое имя:

```
>>> don = Duck('Donald')
>>> don.name
inside the getter
'Donald'
>>> don.name = 'Donna'
inside the setter
>>> don.name
inside the getter
'Donna'
```

Во втором методе вы добавляете декораторы и заменяете имена методов `get_name` и `set_name` на `name`:

- `@property` — размещается перед геттером;
- `@name.setter` — 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 получить доступ к `name`, как будто это атрибут:

```
>>> fowl = Duck('Howard')
>>> fowl.name
inside the getter
'Howard'
>>> fowl.name = 'Donald'
inside the setter
>>> fowl.name
inside the getter
'Donald'
```

Если кто-то и догадается, что мы назвали наш атрибут hidden_name, считать и записать его он все равно может непосредственно как fowl.hidden_name. В подразделе «Искажение имен для безопасности» далее в этой главе вы увидите особый способ Python, позволяющий скрыть имена атрибутов.

Свойства для вычисляемых значений

В предыдущих примерах мы использовали свойство `name`, чтобы обратиться к отдельному атрибуту (в нашем случае `hidden_name`), который хранится внутри объекта.

Свойство может возвращать и *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 если бы это был атрибут, подобный `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 рассмотренном ранее примере с классом `Duck` мы назвали 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 так, как нужно:

```
>>> fowl = Duck('Howard')
>>> fowl.name
inside the getter
'Howard'
>>> fowl.name = 'Donald'
inside the setter
>>> fowl.name
inside the getter
'Donald'
```

Выглядит неплохо. 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 полностью закрытым: Python искажает имя, чтобы внешний код не наткнулся на этот атрибут. Если вам интересно и вы обещаете никому не рассказывать[1], 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 неидеальна, искажение имен препятствует случайному или преднамеренному доступу к атрибуту.

Атрибуты классов и объектов

Вы можете присвоить атрибуты классам, и они будут унаследованы их объектами-потомками:

```
>>> class Fruit:
...     color = 'red'
...
```

[1] Вы умеете хранить секреты? Я, кажется, нет.

```
>>> blueberry = Fruit()
>>> Fruit.color
'red'
>>> blueberry.color
'red'
```

Но если вы измените значение атрибута в объекте-потомке, атрибут класса не изменится:

```
>>> blueberry.color = 'blue'
>>> blueberry.color
'blue'
>>> Fruit.color
'red'
```

Если позже измените атрибут класса, на существующих объектах-потомках это не отразится:

```
>>> Fruit.color = 'orange'
>>> Fruit.color
'orange'
>>> blueberry.color
'blue'
But it will affect new ones:
>>> new_fruit = Fruit()
>>> new_fruit.color
'orange'
```

Типы методов

Одни методы являются частью самого класса, другие — частью объектов, созданных из этого класса, а какие-то не являются ни тем ни другим.

- Если перед методом нет декоратора, то это *метод объекта* и его первым аргументом должен быть `self`, который ссылается на объект.
- Если у метода есть декоратор `@classmethod`, то это *метод класса* и его первым аргументом должен быть `cls` (имя может быть любым, кроме зарезервированного слова `class`), который ссылается на класс.
- Если у метода есть декоратор `@staticmethod`, то это *статический метод* и его первым аргументом будет не объект и не класс.

В следующих подразделах мы рассмотрим эти вопросы более детально.

Методы объектов

Когда вы видите начальный аргумент `self` в методах внутри определения класса, вы имеете дело с *методом объекта*: такие методы вы обычно пишете, когда создаете собственный класс. Первый параметр метода экземпляра — это `self`, Python передает объект методу, когда вы его вызываете. Это единственные методы, которые мы уже рассмотрели.

Методы классов

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. По традиции этот параметр называется `cls`, поскольку слово `class` зарезервировано и не может быть использовано в данной ситуации. 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 применить вызов `A.count`.

Статические методы

Третий тип методов не влияет ни на классы, ни на объекты: он находится внутри класса только для удобства, чтобы не располагаться где-то отдельно. Это *статический метод*, которому предшествует декоратор `@staticmethod`, не имеющий в качестве начального параметра ни `self`, ни класс `class`. Рассмотрим пример, который является рекламой 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. И это замечательно.

Утиная типизация

В Python имеется также реализация *полиморфизма* — это значит, что одна операция может быть произведена над разными объектами, основываясь на имени метода и его аргументах, 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 переопределяли их методы `__init__()`. Далее Python автоматически вызывает метод `__init__()` родительского класса `Quote`, чтобы сохранить переменные объекта `person` и `words`. Таким образом, мы можем получить доступ к атрибуту `self.words` в объектах, созданных с помощью подклассов `QuestionQuote` и `ExclamationQuote`.

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 вызывать методы `who()` и `says()` для любых объектов, в том числе и для этих методов. Определим класс `BabblingBrook`, который не имеет никакого отношения к нашим охотнику и его жертвам (потомкам 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
```

Благодаря старинной поговорке «Если нечто выглядит как утка, плавает как утка и крякает как утка, то, скорее всего, это и есть утка» подобное поведение иногда называют *утиной типизацией* (рис. 10.1). Кто мы такие, чтобы спорить с мудрецом, рассуждающим об утках?

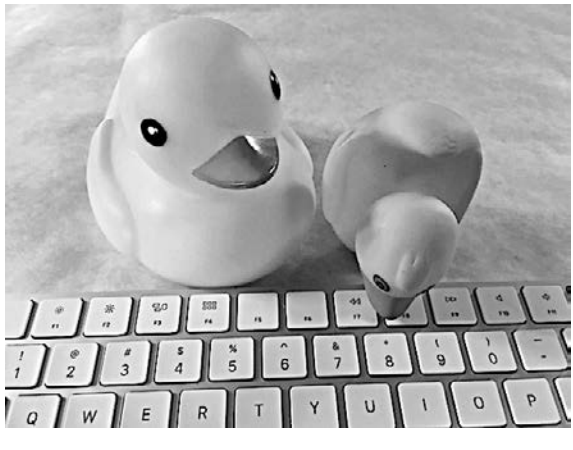

Рис. 10.1. Утиная типизация — это не значит печатать текст, сложив губы уточкой

Магические методы

Теперь вы можете создавать и использовать простые объекты. То, что мы изучим в этом разделе, может вас удивить — в хорошем смысле этого слова.

Когда вы пишете что-то вроде `a = 3 + 8`, откуда целочисленные объекты со значениями `3` и `8` узнают, как реализовать операцию `+`? А если вы введете конструкцию `name = "Daffy" + " " + "Duck"`, как Python узнает, что оператор `+` теперь означает конкатенацию строк? И что подскажет переменным `a` и `name`, как применить `=`, чтобы получить результат? Вы можете получить доступ к этим операторам, используя *специальные методы* Python (или, что звучит более драматично, *магические методы*).

Имена этих методов начинаются и заканчиваются двойными подчеркиваниями (`__`). Почему? Как правило, такие имена не выбирают в качестве имен для переменных. Вы уже видели один такой метод: `__init__()` инициализирует только что созданный объект с помощью описания его класса и любых аргументов, которые передаются в этот метод. Вы также видели (см. подраздел «Искажение имен для безопасности»), как использование в именах двойного нижнего подчеркивания помогает исказить имена атрибутов класса и его методов.

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 `equals()` — обычный метод. `self.text` — это текстовая строка, которую содержит объект класса `Word`. Метод `equals()` сравнивает ее с текстовой строкой, содержащейся в объекте `word2` (другом объекте 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()` для преобразования строки в нижний регистр и сравнения. Но было бы неплохо, если б мы могли просто сказать `first == second`, как в случае встроенных типов Python. Реализуем такую возможность. 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, работает ли это:

```
>>> 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. 10.1 и 10.2 приведены имена самых полезных магических методов.

Таблица 10.1. 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 10.2. 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 такие математические операторы, как + (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. 10.3.

Таблица 10.3. Другие магические методы

__str__(self)	str(self)
__repr__(self)	repr(self)
__len__(self)	len(self)

Часто, помимо `__init__()`, вы будете применять и метод `__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 5. 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 специальных методах, обратитесь к документации Python (http://bit.ly/pydocs-smn).

Агрегирование и композиция

Наследование может сослужить хорошую службу, когда вы хотите, чтобы класс-потомок бóльшую часть времени вел себя как родительский класс (потомок является родителем). Возможность создавать иерархии наследования довольно заманчива, но иногда *композиция* или *агрегирование* имеет больше смысла. В чем разница? При композиции один объект является частью другого. Утка *является* птицей (наследование), но *имеет* хвост (композиция). 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', self.bill.description, 'bill and a',
                self.tail.length, 'tail')
...
>>> a_tail = Tail('long')
>>> a_bill = Bill('wide orange')
>>> duck = Duck(a_bill, a_tail)
>>> duck.about()
This duck has a wide orange bill and a long tail
```

Агрегирование выражает более свободные отношения между объектами: один из них *использует* другой, но оба они существуют независимо друг от друга. Утка *использует* озеро, но не является его частью.

Когда использовать объекты, а когда — что-то другое

Рассмотрим несколько рекомендаций, которые помогут вам понять, где лучше разместить свой код и данные — в классе, в модуле (их мы рассмотрим в главе 11) или в чем-то совершенно ином.

- Объекты наиболее полезны, когда вам нужно иметь несколько отдельных экземпляров с одинаковым поведением (методы), но разным внутренним состоянием (атрибуты).
- Классы, в отличие от модулей, поддерживают наследование.

- Если вам нужен только один объект, модуль может быть лучшим выбором. Независимо от того, сколько обращений к модулю имеется в программе, загрузится только одна копия. (Программистам на Java и C++: можете использовать модули в Python как *синглтоны*.)
- Если у вас есть несколько переменных, которые содержат разные значения и могут быть переданы в качестве аргументов 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()`. Правда, по мере добавления новых ключей и функций разбираться со всем этим станет трудно. Поэтому более предпочтительно определить класс `Image` с атрибутами `size` или `color` и методами `scale()` и `transform()`. Тогда все данные и методы для работы с цветными изображениями будут определены в одном месте.
- Используйте самые простые решения. 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. (Встроенные типы данных — ваши друзья.) Используйте больше чисел, строк, кортежей, списков, множеств, словарей. Взгляните также на библиотеку `collections`, особенно на класс `deque`».
- Более новой альтернативой является *класс данных*, он рассматривается далее в этой главе.

Именованные кортежи

Поскольку Гвидо только что упомянул их, а я еще не говорил, самое время рассмотреть *именованные кортежи*. Именованный кортеж — это подкласс кортежей, с помощью которых вы можете получить доступ к значениям по имени (используя конструкцию `.`*`имя`*) и позиции (используя конструкцию `[`*`смещение`*`]`).

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: чтобы их использовать, вам понадобится загрузить отдельный модуль. 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 из словаря:

```
>>> parts = {'bill': 'wide orange', 'tail': 'long'}
>>> duck2 = Duck(**parts)
>>> duck2
Duck(bill='wide orange', tail='long')
```

В предыдущем коде обратите внимание на конструкцию **parts. Это *аргумент — ключевое слово*. Он извлекает ключи и значения словаря parts и передает их в качестве аргументов в Duck(), что приводит к тому же эффекту, как и:

```
>>> 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 определить 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: ' Duck' object has no attribute 'color'
```

Вспомним плюсы использования именованных кортежей.

- 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, используя точки, а не квадратные скобки, характерные для словарей.
- Вы можете использовать их как ключ словаря.

Классы данных

Многие люди создают объекты для хранения данных (с помощью атрибутов объектов), а не для задания определенного поведения (с помощью методов). Вы уже видели, как именованные кортежи могут стать альтернативным хранилищем данных. В Python 3.7 появились *классы данных*.

Перед вами простой старый объект, имеющий один атрибут `name`:

```
>> class TeenyClass():
...     def __init__(self, name):
...         self.name = name
...
>>> teeny = TeenyClass('itsy')
>>> teeny.name
'itsy'
```

Реализация той же функциональности с помощью классов данных будет выглядеть несколько иначе:

```
>>> from dataclasses import dataclass
>>> @dataclass
... class TeenyDataClass:
...     name: str
...
>>> teeny = TeenyDataClass('bitsy')
>>> teeny.name
'bitsy'
```

Помимо декоратора `@dataclass`, вам нужно определить атрибуты класса с помощью *аннотаций переменных* (https://www.python.org/dev/peps/pep-0526/). Это выглядит как *`имя: тип`* или *`имя: тип = значение`*, например `color: str` или `color: str = "red"`. Типом может быть любой тип объектов в Python, включая созданные вами классы, а не только встроенные, такие как `str` или `int`.

Когда вы создаете объект класса данных, вы предоставляете аргументы в том порядке, в котором они указаны в классе, или используете именованные аргументы, передавая их в любом порядке:

```
>>> from dataclasses import dataclass
>>> @dataclass
... class AnimalClass:
...     name: str
...     habitat: str
...     teeth: int = 0
...
>>> snowman = AnimalClass('yeti', 'Himalayas', 46)
>>> duck = AnimalClass(habitat='lake', name='duck')
>>> snowman
AnimalClass(name='yeti', habitat='Himalayas', teeth=46)
>>> duck
AnimalClass(name='duck', habitat='lake', teeth=0)
```

В классе `AnimalClass` определено значение по умолчанию для атрибута `teeth`, поэтому вам не нужно предоставлять его значение при создании объекта `duck`.

Вы можете обращаться к атрибутам этого объекта точно так же, как и к атрибутам других объектов:

```
>>> duck.habitat
'lake'
>>> snowman.teeth
46
```

О классах данных можно говорить еще долго. Обратитесь к этому руководству (https://realpython.com/python-data-classes/) или прочтите официальную (довольно объемную) документацию (https://www.python.org/dev/peps/pep-0557/).

attrs

Вы уже видели, как создаются классы и как в них добавляются атрибуты, а также как много текста для этого требуется набрать: надо определить метод-инициализатор `__init__()`, присвоить значения его аргументов атрибутам класса и создать методы, в имени которых присутствует двойное подчеркивание, например `__str__()`. Именованные кортежи и классы данных предлагают альтернативное решение, которое можно встретить в стандартной библиотеке, — если вам нужно создать простую коллекцию данных, проще всего воспользоваться этими вариантами.

В статье *The One Python Library Everyone Needs* (https://glyph.twistedmatrix.com/2016/08/attrs.html) сравниваются простые классы, именованные кортежи и классы данных. В ней по множеству причин рекомендуется использовать сторонний пакет `attrs` (https://oreil.ly/Rdwlx), который позволяет меньше набирать, меньше проверять данные и т. д. Взгляните сами и решите для себя, что вам больше нравится — эта библиотека или встроенные решения.

Читайте далее

В следующей главе мы обратимся к более высокому уровню структурирования кода — к *модулям* и *пакетам*.

Упражнения

10.1. Создайте класс `Thing`, не имеющий содержимого, и выведите его на экран. Затем создайте объект `example` этого класса и также выведите его. Совпадают ли выведенные значения?

10.2. Создайте новый класс с именем `Thing2` и присвойте переменной `letters` значение `'abc'`. Выведите на экран значение `letters`.

10.3. Создайте еще один класс, который, конечно же, называется `Thing3`. В этот раз присвойте значение `'xyz'` атрибуту объекта `letters`. Выведите на экран зна-

чение атрибута `letters`. Понадобилось ли вам создавать объект класса, чтобы сделать это?

10.4. Создайте класс `Element`, имеющий атрибуты объекта `name`, `symbol` и `number`. Создайте объект этого класса со значениями `'Hydrogen'`, `'H'` и `1`.

10.5. 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.

10.6. 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.

10.7. 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)`.

10.8. Модифицируйте класс `Element`, сделав атрибуты `name`, `symbol` и `number` приватными. Определите свойство получателя для каждого атрибута, возвращающее его значение.

10.9. 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.

10.10. 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.

ГЛАВА 11
Модули, пакеты и программы

Вы уже прошли путь от встроенных типов данных до более крупных структур данных и кода. В этой главе вы доберетесь до самого главного и узнаете, как писать работающие и объемные программы на Python. Вы напишете свои *модули*, а также научитесь использовать чужие, взятые из *стандартной библиотеки* Python и других источников.

Текст этой книги организован в виде иерархической структуры: слова, предложения, абзацы и главы. Иначе читать его стало бы невозможно уже через пару страниц[1]. Код имеет подобную организацию: типы данных можно сравнить со словами, операторы и выражения — с предложениями, функции можно представить абзацами, а модули — главами. Продолжу аналогию: когда я говорю, что какую-то тему более подробно мы рассмотрим, например, в главе 8, в программировании это была бы отсылка к коду другого модуля.

Модули и оператор import

Сейчас мы будем создавать код из нескольких файлов и применять его. *Модуль* — это просто файл, содержащий код Python. Вам не нужно делать ничего особенного — любой код, написанный на Python, может быть использован другими как модуль.

На код других модулей мы ссылаемся с помощью оператора `import`. Это делает код и переменные импортированного модуля доступными для вашей программы.

Импортируем модуль

Простейший вариант использования оператора `import` выглядит как `import` *`модуль`*, где *`модуль`* — это имя другого файла Python без расширения `.py`.

Предположим, вы с друзьями хотите съесть на обед что-то из фастфуда, но не желаете долго выбирать, поскольку все равно последнее слово останется за тем, кто будет громче всех кричать. Пусть выбирает компьютер! Напишем модуль всего с одной функцией, которая будет возвращать случайное блюдо, и основную программу, которая вызовет эту функцию и выведет результат на экран. Модуль (`fast.py`) показан в примере 11.1.

---

[1] Как минимум читать было бы гораздо сложнее.

Пример 11.1. fast.py

```
from random import choice

places = ["McDonalds", "KFC", "Burger King", "Taco Bell",
          "Wendys", "Arbys", "Pizza Hut"]

def pick():  # смотрите строку документации
    """Return random fast food place"""
    return choice(places)
```

А в примере 11.2 показана основная программа, которая его импортирует (назовем ее `lunch.py`).

Пример 11.2. lunch.py

```
import fast

place = fast.pick()
print("Let's go to", place)
```

Если вы поместите оба этих файла в один каталог и укажете Python запустить файл `lunch.py` в качестве основной программы, он обратится к модулю `fast` и запустит его функцию `pick()`. Мы написали эту версию функции `pick()` так, чтобы она возвращала случайный результат из списка строк. Таким образом, основная программа выведет:

```
$ python lunch.py
Let's go to Burger King
$ python lunch.py
Let's go to Pizza Hut
$ python lunch.py
Let's go to Arbys
```

Мы использовали импортирование дважды.

- Основная программа `lunch.py` импортировала наш новый модуль `fast`.
- Файл модуля `fast.py` импортировал функцию `choice` из стандартного модуля Python `random`.

Мы также использовали импортирование двумя разными способами.

В первом случае мы импортировали модуль `fast` целиком, но для этого нам нужно было поставить `fast` в качестве префикса для `pick()`. Если мы ставим префикс `fast` перед именем вызываемой функции, то все содержимое файла `fast.py` после оператора `import` становится доступным основной программе. Уточняя содержимое модуля с помощью его имени, мы избегаем возникновения любых неприятных конфликтов имен: в каком-то другом модуле также может быть функция `pick()`, 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 модуль `fast.py` так, как это показано в примере 11.3, импортировав модуль `random` внутри функции `pick()`, а не в начале файла.

Пример 11.3. fast2.py

```
places = ["McDonalds", "KFC", "Burger King", "Taco Bell",
          "Wendys", "Arbys", "Pizza Hut"]

def pick():
    import random
    return random.choice(places)
```

Как и вообще в программировании, лучше выбирать стиль, который кажется вам наиболее понятным. Имя функции со стоящим перед ним именем модуля `random.choice` использовать безопаснее, однако в таком случае придется набирать немного больше текста.

Вам следует рассмотреть возможность импортировать код вне функции, если импортированный код может быть использован более одного раза, и изнутри функции, если вы знаете, что его применение будет лимитировано. Кто-то предпочитает размещать все операторы `import` в верхней части файла, чтобы явно обозначить все зависимости кода. Оба варианта работают.

Импортируем модуль с другим именем

В нашей основной программе `lunch.py` мы вызывали `import fast`. Но что, если:

- у вас есть другой модуль с таким же именем;
- вы хотите использовать более короткое или простое имя;
- ваши пальцы прищемило дверью и вы хотите набирать меньше текста?

В такой ситуации можно импортировать, используя *псевдоним*, как показано в примере 11.4. Используем псевдоним `f`.

Пример 11.4. fast3.py

```
import fast as f
place = f.pick()
print("Let's go to", place)
```

Импортируем только самое необходимое

С помощью Python можно импортировать одну или несколько частей модуля. Со второй ситуацией вы уже имели дело: из модуля `random` нам нужна была только функция `choice()`.

Как и с модулями, вы можете дать псевдоним каждой части, которую импортируете.

Теперь несколько раз перепишем файл `lunch.py`. Сначала импортируем функцию `pick()` из модуля `fast` с ее исходным именем (пример 11.5).

Пример 11.5. fast4.py

```
from fast import pick
place = pick()
print("Let's go to", place)
```

Теперь импортируем ее под именем `who_cares` (пример 11.6).

Пример 11.6. fast5.py

```
from fast import pick as who_cares
place = who_cares()
print("Let's go to", place)
```

Пакеты

Мы проделали путь от отдельных строк кода до многострочных функций, автономных программ и нескольких модулей в одном каталоге. Если модулей у вас немного, можно разместить их в одной папке.

Для того чтобы еще больше промасштабировать приложения Python, вы можете организовать модули в иерархии файлов и модулей, которые называют `пакетами`. Пакет — это обычный подкаталог, содержащий файлы с расширением `.py`. Внутри каталогов можно создавать другие подкаталоги, которые, в свою очередь, могут иметь свои подкаталоги.

Мы только что написали модуль для выбора места, где можно поесть фастфуд. Давайте добавим похожий модуль для выдачи жизненного совета. Итак, создадим еще одну основную программу с названием `questions.py` в текущем каталоге. Теперь создадим подкаталог с именем `choices` и поместим туда два модуля — `fast.py` и `advice.py`. Каждый модуль имеет функцию, которая возвращает строку.

Основная программа `questions.py` будет содержать дополнительное утверждение `import` и еще одну строку (пример 11.7).

Пример 11.7. questions.py

```
from sources import fast, advice

print("Let's go to", fast.pick())
print("Should we take out?", advice.give())
```

Утверждение `from sources` заставляет Python искать каталог с именем `sources`, начиная с вашего текущего каталога. Внутри каталога `sources` он ищет файлы `fast.py` и `advice.py`.

Код первого модуля `choices/fast.py` не изменился, просто файл переместился в каталог `choices` (пример 11.8).

Пример 11.8. choices/fast.py

```
from random import choice

places = ["McDonalds", "KFC", "Burger King", "Taco Bell",
          "Wendys", "Arbys", "Pizza Hut"]

def pick():
    """Return random fast food place"""
    return choice(places)
```

Второй модуль `choices/advice.py` новый, но работает практически так же, как и его собрат, посвященный фастфуду (пример 11.9).

Пример 11.9. choices/advice.py

```
from random import choice

answers = ["Yes!", "No!", "Reply hazy", "Sorry, what?"]

def give():
    """Return random advice"""
    return choice(answers)
```

Если версия вашего Python ниже, чем 3.3, понадобится еще кое-что сделать в подкаталоге sources, чтобы он стал пакетом Python, — создать файл с именем __init__.py. Это может быть даже пустой файл, но для более старых версий Python его присутствие необходимо, чтобы содержащий его каталог считался пакетом. (Это еще один распространенный вопрос на собеседованиях.)

Запустите основную программу `questions.py` (из вашего текущего каталога, а не из `sources`), чтобы увидеть, что произойдет:

```
$ python questions.py
Let's go to KFC
Should we take out? Yes!
$ python questions.py
Let's go to Wendys
Should we take out? Reply hazy
$ python questions.py
Let's go to McDonalds
Should we take out? Reply hazy
```

Путь поиска модуля

Я только что сказал, что Python просматривает ваш текущий каталог для поиска подкаталога `choices` и его модулей. На самом деле он выполняет поиск не только там, и вы можете сами выбирать места поиска нужных файлов. Ранее мы импортировали функцию `choice()` из модуля стандартной библиотеки `random`. Его не было в текущем каталоге, поэтому Python надо было искать и в другом месте.

Чтобы узнать, где интерпретатор Python ищет файлы, импортируйте стандартный модуль `sys` и используйте его список `path`, в котором содержатся имена каталогов и файлы ZIP-архивов.

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
```

Эта начальная пустая строка вывода является пустой строкой '', обозначающей текущий каталог. Если строка '' находится первой в sys.path, Python сначала выполнит поиск в текущем каталоге, когда вы попробуете что-то импортировать: import fast выглядит как fast.py. Python обычно действует именно так. Кроме того, если создать подкаталог sources и поместить туда файлы с кодом, их можно импортировать командами import sources и from sources import fast.

Использован будет 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 элементу random стандартной библиотеки.

Вы можете изменить путь, по которому производится поиск, внутри вашего кода. Предположим, вы хотите, чтобы Python искал файлы в каталоге /my/modules перед тем, как обратиться к другим местам:

```
>>> import sys
>>> sys.path.insert(0, "/my/modules")
```

Относительный и абсолютный импорт

К этому моменту в наших примерах мы импортировали собственные модули из:

- текущего каталога;
- подкаталога choices;
- стандартной библиотеки Python.

Этот подход хорошо работает до тех пор, пока у вас нет локального модуля с тем же именем, что и у стандартного модуля. Какой именно из них вам нужен?

Python поддерживает *абсолютный* и *относительный* импорт. В рассмотренных примерах вы видели абсолютный импорт. Если вы введете команду import rougarou, то для каждого указанного каталога Python будет искать файл с именем rougarou.py (модуль) или каталог с именем rougarou (пакет).

- Если файл rougarou.py находится в том же каталоге, что и вызывающая сторона, вы можете импортировать его относительно текущей локации с помощью команды from . import rougarou.
- Если он находится в каталоге более высокого уровня: from .. import rougarou.
- Если находится в каталоге того же уровня с именем creatures: from ..creatures import rougarou.

Нотации . и .. были позаимствованы из ОС семейства Unix — там эти сокращения используются для обозначения *текущего* и *родительского каталогов*.

Подробнее о проблемах импорта в Python вы можете прочитать в статье *Traps for the Unwary in Python's Import System* (http://python-notes.curiousefficiency.org/en/latest/python_concepts/import_traps.html).

Пакеты пространств имен

Вы уже знаете, что можно упаковать модули Python как:

- единый *модуль* (файл с расширением .py);
- *пакет* (каталог, содержащий модули и, возможно, другие пакеты).

Вы также можете разбить пакет на несколько каталогов с помощью *пакетов пространств имен*. Предположим, вы ходите создать пакет `critters`, который будет содержать модули Python для каждого опасного существа (реального или вымышленного). Кроме того, в модулях будут содержаться вспомогательная информация и заметки по выживанию. Со временем пакет может разрастись, поэтому вы решаете разделить модули по географическому расположению существ. Один из вариантов решения проблемы — создание подпакета для каждой локации и перемещение в него существующих файлов модулей `.py`. Однако это может нарушить работу других модулей, которые импортируют данные файлы. Вместо этого можно сделать следующее:

- создать новые каталоги для каждой локации на уровень выше пакета `critters`;
- создать «двоюродные» `critters`-каталоги ниже этих новых «родителей»;
- переместить существующие модули в соответствующие каталоги.

Этот подход нужно проиллюстрировать. Предположим, мы начали работу с такой раскладкой файлов:

```
critters
 rougarou.py
 wendigo.py
```

Обычные команды импорта для модулей будут выглядеть следующим образом:

```
from critters import wendigo, rougarou
```

Теперь, если мы решим выделить локации `north` и `south`, файлы будут размещаться вот так:

```
north
 critters
   wendigo.py
south
 critters
   rougarou.py
```

Если оба каталога, `north` и `south`, будут находиться в вашем пути поиска модуля, вы сможете импортировать модули так же, как если бы они все еще находились в пакете, состоящем из одного каталога:

```
from critters import wendigo, rougarou
```

Модули против объектов

В каких случаях нужно размещать код в модуле, а в каких — в объекте?

У модуля и объекта много общего. Объект или модуль с именем `thing`, содержащий переменную с именем `stuff`, позволяет вам получить доступ к этой переменной с помощью конструкции `thing.stuff`. Переменная `stuff` может быть определена при создании модуля или класса, или же ее значение может быть присвоено позже. Все классы, функции и глобальные переменные модуля доступны и за его пределами.

Объекты могут использовать свойства и двойные нижние подчеркивания (`__ …`) в именах для того, чтобы спрятать свои данные или управлять доступом к ним.

Это значит, что вы можете сделать следующее:

```
>>> import math
>>> math.pi
3.141592653589793
>>> math.pi = 3.0
>>> math.pi
3.0
```

Неужели мы только что испортили вычисления для всех, кто пользуется этим компьютером? Да! На самом деле нет, я шучу[1]. Это действие не повлияло на модуль `math`. Вы изменили только значение `pi` для копии модуля `math`, которая импортируется вашей вызывающей программой, и все улики вашего преступления исчезнут по ее завершении.

Для любого импортированного вами модуля создается только одна копия, даже если вы импортируете его несколько раз. Этим можно пользоваться для сохранения глобальных объектов, представляющих интерес для вашей программы: так же как и с классом, который тоже имеет всего одну копию, хотя из него можно создать много объектов.

Достоинства стандартной библиотеки 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 во избежание разрастания ядра языка. Нередко, когда вы собираетесь писать код, сначала стоит проверить, нет ли уже стандартного модуля, который делает то, что вы хотите. Python также предоставляет авторитетную документацию для модулей наряду с руководством для пользователей (http://docs.python.org/3/library). Сайт Дага Хеллмана 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 вы познакомитесь со множеством стандартных модулей, предназначенных для работы с сетью, системами, базами данных и т. д. В этом разделе я говорю о стандартных модулях, которые имеют более общие варианты использования.

Обрабатываем отсутствующие ключи с помощью функций setdefault() и defaultdict()

Вы уже знаете, что попытка получить доступ к словарю с помощью несуществующего ключа генерирует исключение. Избежать этого помогает функция словаря `get()` для возврата значения по умолчанию. Функция `setdefault()` похожа на

[1] Или не шучу? Муа-ха-ха!

функцию `get()`, но кроме того, она 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 попытаемся 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 возвращать значение `0`:

```
>>> from collections import defaultdict
>>> periodic_table = defaultdict(int)
```

Теперь любое отсутствующее значение будет заменяться целым числом `int` со значением `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()` вызывается 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 `[]`, `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 внутри вызова:

```
>>> bestiary = defaultdict(lambda: 'Huh?')
>>> bestiary['E']
'Huh?'
```

Использовать `int` — это один из способов создать собственный счетчик:

```
>>> 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
```

Если бы в предыдущем примере `food_counter` был обычным словарем, а не `defaultdict`, Python генерировал бы исключение всякий раз при попытке увеличить элемент словаря `food_counter[food]`, ведь тот 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 справляется с задачей из предыдущего примера, и не только с ней:

```
>>> 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 только 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})
```

Создадим новый список с именем `lunch` и счетчик соответственно с именем `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 нетрудно догадаться, счетчики можно вычитать друг из друга с помощью оператора `-`. Например, «что мы будем есть на завтрак, но не на обед?».

```
>>> breakfast_counter – lunch_counter
Counter({'spam': 3})
```

О'кей. А 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 8, 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 наименьшим количеством. Это имеет смысл: на завтрак было только одно яйцо, поэтому указанное (наименьшее) 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()

Перед вами пример, запущенный с помощью интерпретатора Python 2:

```
>>> quotes = {
...     'Moe': 'A wise guy, huh?',
...     'Larry': 'Ow!',
...     'Curly': 'Nyuk nyuk!',
...     }
>>> for stooge in quotes:
...  print(stooge)
...
Larry
Curly
Moe
```

Начиная с версии Python 3.7, словари сохраняют очередность ключей, в которой они были добавлены. В ранних версиях можно использовать OrderedDict, поскольку порядок ключей в обычных словарях не определен. Примеры из этого подраздела актуальны только в том случае, если версия вашего Python ниже 3.7.

Функция OrderedDict() запоминает порядок, в котором добавлялись ключи, и возвращает их в том же порядке с помощью итератора. Попробуем создать словарь 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 двусторонняя очередь, которая имеет функции как стека, так и очереди. Она полезна, когда нужно добавить или удалить элементы с любого конца последовательности. В следующем примере мы будем двигаться

с обоих концов слова к его середине, чтобы проверить, является ли оно палиндромом. Функция `popleft()` удаляет крайний слева элемент и возвращает его, функция `pop()` удаляет крайний справа элемент и возвращает его. Вместе они движутся с концов слова к его середине. Эти действия продолжатся до тех пор, пока совпадают крайние символы 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 проще проиллюстрировать работу `deque`. Если же вы действительно хотите создать программу для определения палиндромов, гораздо удобнее сравнивать строку с ее «перевернутой» копией. В Python строковой функции `reverse()` не существует, но можно записать строку в обратном порядке с помощью слайса, 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 модуля itertools

Модуль `itertools` (https://docs.python.org/3/library/itertools.html) содержит особые функции итератора. Все они возвращают один элемент при каждом вызове из цикла `for… in` и запоминают 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 гораздо больше функций. Некоторые из них, предназначенные, в частности, для комбинаций и перестановок, могут сэкономить много времени, если это будет нужно.

Красиво выводим данные на экран с помощью функции pprint()

Во всех наших примерах использовалась функция `print()` (или просто имя переменной в интерактивном интерпретаторе) для вывода информации на экран. Иногда прочитать результаты бывает трудно. Нам нужен `pretty printer` (красивый принтер), такой как `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!'}
```

Работаем со случайными числами

В начале этой главы мы воспользовались функцией `random.choice()`. Она возвращает значение из последовательности (списка, кортежа, словаря или строки), которая была передана в качестве аргумента:

```
>>> from random import choice
>>> choice([23, 9, 46, 'bacon', 0x123abc])
1194684
>>> choice( ('a', 'one', 'and-a', 'two') )
'one'
>>> choice(range(100))
68
>>> choice('alphabet')
'l'
```

Используем функцию `sample()` для того, чтобы получить больше одного значения за один раз:

```
>>> from random import sample
>>> sample([23, 9, 46, 'bacon', 0x123abc], 3)
[1194684, 23, 9]
>>> sample(('a', 'one', 'and-a', 'two'), 2)
['two', 'and-a']
>>> sample(range(100), 4)
```

```
[54, 82, 10, 78]
>>> sample('alphabet', 7)
['l', 'e', 'a', 't', 'p', 'a', 'b']
```

Чтобы получить случайное целое число из любого диапазона, можно обратиться к функциям `choice()` или `sample()`, передав в них возвращаемое значение функции `range()`. Можно воспользоваться также функциями `randint()` или `randrange()`:

```
>>> from random import randint
>>> randint(38, 74)
71
>>> randint(38, 74)
60
>>> randint(38, 74)
61
```

Функция `randrange()`, как и функция `range()`, имеет аргументы для начального значения (включительно), конечного (не включительно), а также для необязательного целочисленного параметра шага:

```
>>> from random import randrange
>>> randrange(38, 74)
65
>>> randrange(38, 74, 10)
68
>>> randrange(38, 74, 10)
48
```

Итак, получим случайное вещественное число (число с плавающей точкой) между `0.0` и `1.0`:

```
>>> from random import random
>>> random()
0.07193393312692198
>>> random()
0.7403243673826271
>>> random()
0.9716517846775018
```

Нужно больше кода

Иногда стандартная библиотека не имеет того, что вам нужно, или делает что-то не так, как вы хотите. В таком случае стоит вспомнить о том, что существует целый мир стороннего программного обеспечения Python с открытым исходным кодом. Отлично зарекомендовали себя следующие ресурсы:

- PyPi (известный также как Cheese Shop («Магазин сыра») после старой пародии Monty Python) (http://pypi.python.org/);
- 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 везде в этой книге код Python использует функции стандартных библиотек Python. Но кое-где представлены и внешние пакеты: в главе 1 я упоминал `requests`, а в главе 18 рассматривал его более подробно. В приложении Б показано, как устанавливать стороннее программное обеспечение Python, и дана другая подробная информация.

Читайте далее

Следующая глава посвящена аспектам манипуляции данными в Python. Вы познакомитесь с типами `bytes` и `bytearray`, обработаете символы Unicode в строке, а также научитесь выполнять поиск в строке с помощью регулярных выражений.

Упражнения

11.1. Создайте файл с именем `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()`.

11.2. В интерактивном интерпретаторе импортируйте модуль `zoo` под именем `menagerie` и вызовите его функцию `hours()`.

11.3. Оставаясь в интерпретаторе, импортируйте непосредственно функцию `hours()` из модуля `zoo` и вызовите ее.

11.4. Импортируйте функцию `hours()` под именем `info` и вызовите ее.

11.5. Создайте словарь с именем `plain`, содержащий пары «ключ — значение» `'a': 1`, `'b': 2` и `'c':3`, а затем выведите его на экран.

11.6. Создайте `OrderedDict` с именем `fancy` из пар «ключ — значение», приведенных в упражнении 11.5, и выведите его на экран. Изменился ли порядок ключей?

11.7. 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']`.

ЧАСТЬ II

Python на практике

ГЛАВА 12
Обрабатываем данные

> Если достаточно долго мучить данные, они признаются [в чем угодно].
>
> *Рональд Коуз*

До сих пор мы в основном говорили о самом языке Python — его типах данных, структурах кода, синтаксисе и т. д. Остальная часть книги посвящена применению Python для решения реальных задач.

В текущей главе вы познакомитесь со многими практическими приемами для управления данными: в мире баз данных это иногда называют *манипулированием данными* или ETL (extract/transform/load, то есть «извлечение/преобразование/загрузка»). И хоть в книгах по программированию данная тема обычно не рассматривается отдельно, программистам все равно приходится тратить много времени на то, чтобы привести данные в нужную для своих целей форму.

За последние несколько лет специальность, связанная с *«наукой о данных»*, стала очень популярной. В статье *Harvard Business Review* профессия специалиста, работающего с данными, названа «самой сексуальной в XXI веке». Если речь о востребованности и достойной оплате, тогда ладно, но и рутины в этой профессии более чем достаточно. Наука о данных выходит за пределы обычных требований ETL к базам данных и зачастую включает в себя *машинное обучение* — для того чтобы можно было находить идеи, недоступные человеческому взору. Я начну с рассмотрения простых форматов данных, а затем перейду к самым полезным инструментам науки о данных.

Форматы данных делятся на две категории: *текстовые* и *бинарные*. *Строки* в Python используются для текстовых данных, и как раз в этой главе содержится информация о строках, которую мы пропустили ранее:

- символы *Unicode*;
- *регулярные выражения*.

Затем мы перейдем к бинарным данным и рассмотрим еще два типа, встроенных в Python:

- *bytes* для неизменяемых восьмибитных значений;
- *bytearrays* для изменяемых.

Текстовые строки: Unicode

С основами строк в Python вы познакомились в главе 5. Пришло время более подробно рассмотреть формат Unicode.

Строки в Python 3 представляют собой последовательности символов Unicode, а не массивы байтов. Это, безусловно, самое значительное изменение в языке по сравнению с Python 2.

Все текстовые примеры до сих пор имели формат ASCII (American Standard Code for Information Interchange). Этот формат был определен в 1960-х годах, когда компьютеры были размером с холодильник и считали лишь немногим лучше его.

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, десять цифр, некоторые знаки препинания, символы пробела и непечатаемые символы.

Однако в мире существует больше букв, чем предоставляет формат ASCII. Вы могли заказать в кафе хот-дог, но не Gewürztraminer[1]. Предпринималось множество попыток вместить больше букв и символов в 8 доступных бит, и время от времени вы будете сталкиваться с такими форматами. Вот некоторые из них:

- Latin-1 или ISO 8859-1;
- Windows code page 1252.

Каждый из этих форматов использует все 8 бит, но даже их бывает недостаточно, особенно если нужно воспользоваться неевропейскими языками. *Unicode* — это действующий международный стандарт, определяющий символы всех языков мира плюс математические и другие символы. Еще и эмодзи!

> 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 (12.0) определяется более 137 000 символов, каждый из которых имеет уникальное имя и идентификационный номер. Python 3.8 может работать со всеми этими символами. Они разбиты на восьмибитные наборы, которые называются *плоскостями*. Первые 256 плоскостей называются *основными многоязычными уровнями*. Обратитесь к странице о плоскостях в «Википедии» (https://ru.wikipedia.org/wiki/Плоскость_(Юникод)), чтобы получить более подробную информацию.

---

[1] Название вина в Германии пишется через ü, но теряет эту букву в Эльзасе по пути во Францию.

Строки формата Unicode в Python 3

Если вы знаете Unicode ID или название символа, то можете использовать его в строке Python. Вот несколько примеров.

- Символ `\u` и расположенные за ним *четыре* шестнадцатеричных числа[1] определяют символ, находящийся в одной из 256 многоязычных плоскостей Unicode. Два первых 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 ней следуют *восемь* шестнадцатеричных символов, крайний слева из которых 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="¢"
```

[1] Числа шестнадцатеричной системы счисления, содержащие символы от 0 до 9 и от A до F.

Еще один символ валюты из Unicode:

```
>>>    unicode_test('\u20ac')
value="€", name="EURO SIGN", value2="€"
```

Единственная проблема, с которой вы можете столкнуться, — это ограничения на шрифт для отображения текста. Немногие шрифты имеют изображения для всех символов Unicode — в таком случае вместо отсутствующих символов отображается 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 слово из источника, использующего кодировку UTF-8 (которую мы рассмотрим далее), и вставил его.

Как же нам указать, что последний символ — это é? Если вы посмотрите на индекс символа E, то увидите, что имя E WITH ACUTE, LATIN SMALL LETTER имеет индекс 00E9. Проверим функциями 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 указать символ é и по коду, и по имени:

```
>>> place = 'caf\u00e9'
>>> place
'café'
>>> place = 'caf\N{LATIN SMALL LETTER E WITH ACUTE}'
>>> place
'café'
```

В предыдущем фрагменте вы вставили символ é непосредственно в строку, но также строку можно построить, добавив:

```
>>> 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 Gewürztraminer in a café
```

Строковая функция `len()` считает количество символов в кодировке Unicode, а не байты:

```
>>> len('$')
1
>>>    len('\U0001f47b')
1
```

Зная числовой идентификатор символа в Unicode, можно использовать стандартные функции ord() и chr() для того, чтобы быстро выполнять преобразования между целочисленными идентификаторами и односимвольными строками Unicode:

```
>>> chr(233)
'é'
>>> chr(0xe9)
'é'
>>> chr(0x1fc6)
'□'
```

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 в байты;
- способ *декодировать* байты обратно в строку.

Если бы в Unicode было менее 65 536 символов, мы могли бы хранить ID каждого из них в 2 байтах. Однако символов больше, и если кодировать каждый ID с помощью 3 или 4 байт, объем памяти и дискового пространства, необходимых для обычных текстовых строк, увеличится в четыре раза.

Кен Томпсон и Роб Пайк, чьи имена знакомы работающим с Unix, однажды вечером на салфетке в одной из столовых Нью-Джерси разработали UTF-8 — динамическую схему кодирования. Она использует для символа Unicode от 1 до 4 байт:

- 1 байт для ASCII;
- 2 байта для большинства языков, основанных на латинице (но не кириллице);
- 3 байта для других основных языков;
- 4 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. Гораздо удобнее работать с кодировкой UTF-8, чем постоянно переключаться с одной кодировки на другую.

Если вы создаете строку Python путем копирования и вставки из другого источника, например такого, как веб-страница, убедитесь, что источник был закодирован с помощью UTF-8. Нередко может оказаться, что текст был зашифрован с помощью кодировок Latin-1 или Windows 1252, а это при копировании в строку Python вызывает 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. 12.1.

Таблица 12.1. Кодировки

Имя кодировки	Описание
'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 ее сохранения:

```
>>> 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 данном 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 отличные от UTF-8 кодировки, но тогда вы получите ошибку, если строка Unicode не сможет быть обработана другой кодировкой. Например, если вы используете кодировку ascii, произойдет сбой, 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 избежать возникновения исключений, связанных с кодировкой. Его значение по умолчанию, как можно было видеть в предыдущем примере, равно 'strict': такое значение приводит к исключению UnicodeEncodeError, когда обнаруживается символ, не входящий в кодировку ASCII. Существуют и другие кодировки. Используйте значение 'ignore', чтобы исключить 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, такие как unicode-escape:

```
>>> snowman.encode('ascii', 'backslashreplace')
b'\\u2603'
```

Вы можете использовать этот вариант, если вам нужна печатная версия escape-последовательности Unicode.

Используйте `'xmlcharrefreplace'`, чтобы создавать безопасные для HTML строки:

```
>>> snowman.encode('ascii', 'xmlcharrefreplace')
b'&#9731;'
```

Подробнее о преобразованиях HTML мы поговорим в подразделе «Сущности HTML» на с. 250.

Декодирование

Мы *декодируем* байтовые строки в текстовые строки Unicode. Когда мы получаем текст из какого-то внешнего источника (файлов, баз данных, сайтов, сетевых API и т. д.), он закодирован в виде байтовой строки. Самое сложное — узнать, какая кодировка была использована, чтобы можно было *декодировать* и получить строку Unicode.

Проблема в следующем: ничто в байтовой строке не говорит нам о том, какая кодировка была использована. Я уже упоминал опасность копирования/вставки с сайтов. Вероятно, и вы встречали сайты со странными символами в тех местах, где должны быть обычные 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 5 байт. Первые три такие же, как в 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: по возможности используйте кодировку UTF-8. Она работает и поддерживается везде, она способна выразить любой символ Unicode, и с ее помощью можно быстро кодировать и декодировать.

То, что вы можете указать любой символ Unicode, не значит, что компьютер сможет их все отобразить. Это зависит от шрифта, который вы используете, — вполне вероятно, в результате вы увидите изображение-заполнитель или вовсе ничего. Компания Apple создала шрифт Last Resort Font (https://www.unicode.org/policies/lastresortfont_eula.html) для Unicode Consortium и использует его в своих операционных системах. На странице https://en.wikipedia.org/wiki/Fallback_font в «Википедии» можно найти более подробную информацию. Существует также шрифт Unifont (http://unifoundry.com/unifont/index.html), содержащий символы в диапазоне от \u0000 до \uffff, а также некоторые другие.

Сущности HTML

В Python 3.4 появился еще один способ выполнять преобразования в Unicode и обратно — с помощью *символов-мнемоников* HTML[1]. Этот подход может быть проще, чем поиск имен Unicode, особенно если вы работаете в Интернете:

```
>>> import html
>>> html.unescape("&egrave;")
'è'
```

Такое преобразование также подходит для мнемоников, пронумерованных как в десятичной, так и в шестнадцатеричной системах счисления:

```
>>> import html
>>> html.unescape("&#233;")
'é'
>>> html.unescape("&#xe9;")
'é'
```

[1] По адресу https://html.spec.whatwg.org/multipage/syntax.html#named-character-references вы можете найти таблицу символов-мнемоников в HTML5.

Вы даже можете импортировать именованную сущность, содержащую соответствия, как словарь, и выполнить преобразование самостоятельно. Отбросьте начальный символ '&' в ключе словаря (можно также отбросить и последний символ ';': кажется, сработает в обоих случаях):

```
>>> from html.entities import html5
>>> html5["egrave"]
'è'
>>> html5["egrave;"]
'è'
```

Чтобы выполнить обратное преобразование (из одного символа Python Unicode в символ-мнемоник HTML), сначала нужно получить десятичное значение символа с помощью функции ord():

```
>>> import html
>>> char = '\u00e9'
>>> dec_value = ord(char)
>>> html.entities.codepoint2name[dec_value]
'eacute'
```

Для строк Unicode, содержащих больше одного символа, используйте следующее преобразование, состоящее из двух этапов:

```
>>> place = 'caf\u00e9'
>>> byte_value = place.encode('ascii', 'xmlcharrefreplace')
>>> byte_value
b'caf&#233;'
>>> byte_value.decode()
'caf&#233;'
```

Выражение place.encode('ascii', 'xmlcharrefreplace') вернуло символы ASCII, приведенные к типу bytes (поскольку они были *закодированы*). Следующее выражение byte_value.decode() необходимо для того, чтобы преобразовать переменную byte_value в строку, совместимую с HTML.

Нормализация

Некоторые символы Unicode могут быть представлены более чем одним кодом Unicode. Они выглядят одинаково, но их сравнение даст отрицательный результат, поскольку они содержат разные последовательности байтов. Например, возьмем символ с акутом 'é' в слове 'café'. Создадим односимвольную 'é' несколькими способами:

```
>>> eacute1 = 'é'                              # UTF-8, скопирован
>>> eacute2 = '\u00e9'                         # код Unicode
>>> eacute3 = \                                # имя Unicode
...     '\N{LATIN SMALL LETTER E WITH ACUTE}'
>>> eacute4 = chr(233)               # десятичное байтовое значение
>>> eacute5 = chr(0xe9)              # шестнадцатеричное байтовое значение
>>> eacute1, eacute2, eacute3, eacute4, eacute5
('é', 'é', 'é', 'é', 'é')
>>> eacute1 == eacute2 == eacute3 == eacute4 == eacute5
True
```

Выполним несколько проверок:

```
>>> import unicodedata
>>> unicodedata.name(eacute1)
'LATIN SMALL LETTER E WITH ACUTE'
>>> ord(eacute1)              # как десятичное целое число
233
>>> 0xe9                      # шестнадцатеричное целое число Unicode
233
```

Теперь создадим символ `e` с акутом, объединив простой символ `e` и акут:

```
>>> eacute_combined1 = "e\u0301"
>>> eacute_combined2 = "e\N{COMBINING ACUTE ACCENT}"
>>> eacute_combined3 = "e" + "\u0301"
>>> eacute_combined1, eacute_combined2, eacute_combined3
('é', 'é', 'é'))
>>> eacute_combined1 == eacute_combined2 == eacute_combined3
True
>>> len(eacute_combined1)
2
```

Мы создали символ Unicode из двух других, и он выглядит так же, как и оригинальный символ `'é'`. Но, как говорят на «Улице Сезам», один из них не похож на другой:

```
>>> eacute1 == eacute_combined1
False
```

Если у вас две разные строки Unicode, полученные из разных источников — одна использует `eacute1`, а вторая `eacute_combined1`, — выглядеть они будут одинаково, но поведут себя по-разному.

Исправить это можно с помощью функции `normalize()`, которая содержится в модуле `unicodedata`:

```
>>> import unicodedata
>>> eacute_normalized = unicodedata.normalize('NFC', eacute_combined1)
>>> len(eacute_normalized)
1
>>> eacute_normalized == eacute1
True
>>> unicodedata.name(eacute_normalized)
'LATIN SMALL LETTER E WITH ACUTE'
```

`'NFC'` означает normal form, composed (нормальная форма, составной символ).

Подробная информация

Если хотите узнать больше, эти ссылки будут вам полезны:

- 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).

Текстовые строки: регулярные выражения

В главе 5 мы затронули операции со строками. Вооружившись этой вводной информацией, вы, вероятно, использовали простые «подстановочные» шаблоны в командной строке, такие как команда UNIX `ls *.py`, что означает перечисление всех имен файлов, заканчивающихся 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:

```
>>> import re
>>> result = re.match('You', 'Young Frankenstein')
```

В этом примере строка `'You'` является искомым *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:

```
>>> import re
>>> youpattern = re.compile('You')
```

Далее вы можете выполнить проверку с помощью скомпилированного шаблона:

```
>>> import re
>>> result = youpattern.match('Young Frankenstein')
```

Поскольку в Python с этим часто приходится иметь дело, напомню еще раз: функция match() ищет строку-шаблон только в начале строки-источника, а функция search() ищет шаблон в любом месте источника.

Функция `match()` — не единственный способ сравнить шаблон и источник, существует еще несколько методов (каждый из них мы рассмотрим в следующих подразделах):

- `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.

В большей части примеров регулярных выражений, показанных здесь, используется кодировка ASCII, но строковые функции Python, включая функции для работы с регулярными выражениями, работают с любой строкой Python и любыми символами из кодировки Unicode.

Ищем точное начальное 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
```

А что 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 упоминал в разделе «Новое: I am the Walrus» на с. 89, в Python 3.8 этот пример можно сократить с помощью так называемого *оператора-«моржа»*:

```
>>> import re
>>> source = 'Young Frankenstein'
>>> if m := re.match('Frank', source):
...     print(m.group())
...
```

Теперь используем функцию `search()` для того, чтобы найти шаблон `Frank` в любом месте источника:

```
>>> import re
>>> source = 'Young Frankenstein
```

```
>>> m = re.search('Frank', source)
>>> if m:
...      print(m.group())
...
Frank
```

Изменим шаблон и попробуем найти первое совпадение с помощью функции `match()`:

```
>>> m = re.match('.*Frank', source)
>>> if m:  # функция match возвращает объект
...     print(m.group())
...
Young Frank
```

Кратко объясню, как работает наш новый шаблон `'.*Frank'`:

- символ `.` означает *любой отдельный символ*;
- символ `*` означает *ноль или более 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'`.

Ищем первое совпадение 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 `.*`:

```
>>> import re
>>> source = 'Young Frankenstein'
>>> m = re.search('Frank', source)
>>> if m:    # функция search возвращает объект
...     print(m.group())
...
Frank
```

Ищем все совпадения, используя функцию findall()

В предыдущих примерах мы искали только одно совпадение. Но что, если нужно узнать, сколько раз строка, содержащая один символ `n`, встречается в строке-источнике?

```
>>> import re
>>> source = 'Young Frankenstein'
>>> m = re.findall('n', source)
>>> m   # findall returns a list
['n', 'n', 'n', 'n']
>>> print('Found', len(m), 'matches')
Found 4 matches
```

А что насчет строки 'n', за которой следует любой символ?

```
>>> import re
>>> source = 'Young Frankenstein'
>>> m = re.findall('n.', source)
>>> m
['ng', 'nk', 'ns']
```

Обратите внимание на то, что в совпадения не была записана последняя строка 'n'. Мы должны указать, что символ после 'n' является опциональным, с помощью ?:

```
>>> import re
>>> source = 'Young Frankenstein'
>>> 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:

```
>>> import re
>>> source = 'Young Frankenstein'
>>> 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 предназначен не для простых строк, а для шаблонов:

```
>>> import re
>>> source = 'Young Frankenstein'
>>> m = re.sub('n', '?', source)
>>> m   # sub returns a string
'You?g Fra?ke?stei?'
```

Шаблоны: специальные символы

Многие описания регулярных выражений начинаются с подробной информации о том, как их определить. Я считаю это неправильным. Регулярных выражений не так уж и мало, поэтому в памяти приходится удерживать слишком большое количество деталей. К тому же регулярные выражения используют так много знаков препинания, что выглядят как ругающиеся персонажи мультфильмов.

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 включений предыдущего символа, включая `0`, — это символ *;
- опциональное значение (`0` или `1`) включений предыдущего символа — это символ `?`.

Специальные символы показаны в табл. 12.2.

Таблица 12.2. 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 `string` содержит заранее определенные строковые константы, которые можно использовать для тестирования. Попробуем применить константу `printable`: она содержит 100 печатных 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']
```

Перед вами пробелы в следующем порядке: простой пробел, табуляция, возврат каретки, вертикальная табуляция и перевод страницы.

Регулярные выражения не ограничиваются символами ASCII. Шаблон `\d` совпадает со всем, что в кодировке Unicode считается цифрой, а не только с символами ASCII от 0 до 9. Добавим две не ASCII-буквы в нижнем регистре из `FileFormat.info` (http://www.fileformat.info/info/unicode/category/Ll/list.htm).

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, шаблон нашел только буквы:

```
>>> re.findall('\w', x)
['a', 'b', 'c', 'ê', 'ĕ']
```

Шаблоны: использование спецификаторов

Теперь сделаем «пунктуационную пиццу», используя основные спецификаторы шаблонов для регулярных выражений, показанные в табл. 12.3.

В этой таблице *`expr`* и другие слова, выделенные курсивом, означают любое корректное регулярное выражение.

Таблица 12.3. 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

Шаблон	Совпадения
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.'''
```

Теперь применим разные шаблоны и с помощью регулярных выражений попробуем найти их в строке-источнике.

В следующих примерах в качестве шаблонов я использую простые строки в кавычках. Далее в этом подразделе я покажу, как необработанная строка-шаблон (она начинается с символа r перед первой кавычкой) позволяет избежать некоторых конфликтов между обычными escape-последовательностями в Python и регулярными выражениями. Из соображений безопасности первым аргументом во всех следующих примерах должна быть необработанная строка.

Для начала найдем в любом месте строку `'wish'`:

```
>>> re.findall('wish', source)
['wish', 'wish']
```

Далее найдем 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 escape-последовательность, чтобы найти именно точку:

```
>>> re.findall('fish tonight\.$', source)
['fish tonight.']
```

Начнем с поиска символов w или f, за которыми 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 5, Python использует специальные *escape-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 escape-последовательностей, используя *неформатированные строки* Python, когда определяете строку регулярного выражения. Всегда размещайте символ `r` перед строкой шаблона регулярного выражения, и escape-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, а кортеж, состоящий из них, окажется доступен благодаря вызову `m.groups()`. Так, 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<` *имя* `>` *выр* `)`, он совпадет с выражением *выр* и сохранит совпадение в группе *имя*:

```
>>> 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 последовательности восьмибитных целых чисел с возможными значениями 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`. В этом примере создадим переменную типа `bytes` с именем `the_bytes` и переменную типа `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 из символов \x02 или ASCII, и заканчивается конструкция соответствующим символом кавычки. Python преобразует шестнадцатеричные последовательности или символы ASCII в небольшие целые числа, но показывает байтовые значения, корректно записанные 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`:

```
>>> blist = [1, 2, 3, 255]
>>> the_bytes = bytes(blist)
>>> 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` легко изменяется:

```
>>> blist = [1, 2, 3, 255]
>>> 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 типа `bytearray` Python использует формат `\xxx` для непечатаемых байтов и их эквиваленты ASCII для печатаемых (и заменяет некоторые распространенные escape-последовательности, например `\n` на `\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 показаться запутанным, поскольку перед нами байты (небольшие целые числа), а не символы.

Преобразуем бинарные данные с помощью модуля struct

Как вы уже могли убедиться, Python имеет множество инструментов для манипулирования текстом. Инструментов для работы с бинарными данными гораздо меньше. Стандартная библиотека содержит модуль `struct`, который обрабатывает данные, аналогичные *структурам* в C или C++. С помощью этого модуля можно преобразовать бинарные данные в структуры данных Python и наоборот.

Посмотрим, как он работает с данными из файла с расширением PNG (распространенным форматом изображений, использующимся наряду с GIF и JPEG). Напишем небольшую программу, которая извлекает ширину и высоту изображения из фрагмента данных PNG.

Мы будем использовать логотип издательства O'Reilly — изображение маленького долгопята с глазками-бусинами (рис. 12.1).

Файл этого изображения с расширением PNG доступен в «Википедии» (https://upload.wikimedia.org/wikipedia/en/9/95/O'Reilly_logo.png). Мы не будем рассматривать чтение файлов вплоть до главы 14, поэтому я загрузил этот файл, написал небольшую программу для вывода его значений в байтах и просто распечатал значения первых 30 байт как значения переменной типа `bytes` с именем `data` для примера, размещенного ниже. (Спецификация формата PNG предполагает, что ширина и высота хранятся в первых 24 байтах, поэтому больше данных нам пока и не нужно.)

Рис. 12.1. Долгопят от издательства O'Reilly

```
>>> 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 восьмибайтную последовательность, которая обозначает начало корректного PNG-файла;
- значение переменной `width` извлекается из байтов 16–19, а переменной `height` — из байтов 20–23.

`>LL` — это строка формата, которая указывает функции `unpack()`, как интерпретировать входные последовательности байтов и преобразовывать их в типы данных Python. Рассмотрим ее детальнее:

- символ `>` означает, что целые числа хранятся в формате *big-endian* (прямой порядок байтов);
- каждый символ `L` определяет четырехбайтное целое число типа *unsigned long*.

Вы можете проверить значение каждого четырехбайтного набора напрямую:

```
>>> data[16:20]
b'\x00\x00\x00\x9a'
>>> data[20:24]0x9a
b'\x00\x00\x00\x8d'
```

У целых чисел с прямым порядком байтов (big-endian) главный байт располагается слева. Поскольку значения ширины и длины меньше 255, они умеща-

ются 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 произвести противоположное действие, то есть 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. 12.4 и 12.5 показаны спецификаторы формата для функций `pack()` и `unpack()`.

Спецификаторы порядка байтов (endian) располагаются первыми в строке формата.

Таблица 12.4. Спецификаторы порядка байтов

Спецификатор	Порядок байтов
<	Обратный порядок (Little endian)
>	Прямой порядок (Big endian)

Таблица 12.5. 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

Продолжение ⇩

Таблица 12.5 *(продолжение)*

Спецификатор	Описание	Количество байтов
d	Число с плавающей точкой двойной точности	8
p	Счетчик и символы 1 + *количество*	1 + *количество*
s	Символы	*количество*

Спецификаторы типа следуют за символом, указывающим порядок байтов.

Любому спецификатору может предшествовать число, указывающее *количество*; запись `5B` аналогична записи `BBBBB`.

Вы можете использовать префикс *количества* вместо конструкции `>LL`:

```
>>> struct.unpack('>2L', data[16:24])
(154, 141)
```

Мы использовали разделение `data[16:24]`, чтобы получить интересующие нас байты напрямую. А могли бы использовать спецификатор `x`, чтобы пропустить ненужные нам части:

```
>>> struct.unpack('>16x2L6x', data)
(154, 141)
```

Эта строка означает:

- использовать формат с прямым порядком байтов (>);
- пропустить 16 байт (16x);
- прочитать 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 (https://pypi.org/project/hachoir);
- binio (http://spika.net/py/binio/);
- Kaitai Struct (http://kaitai.io/).

В приложении Б содержатся инструкции о том, как загрузить и установить такие внешние пакеты. В следующем примере используем пакет `construct`. Вот все, что вам необходимо сделать:

```
$ pip install construct
```

Здесь показано, как 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
```

Преобразуем байты/строки с помощью модуля 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 фрагменте мы выведем на экран восьмибайтный заголовок PNG как последовательность шестнадцатеричных значений вместо комбинации символов ASCII и escape-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 обратном направлении это тоже работает:

```
>>> print(binascii.unhexlify(b'89504e470d0a1a0a'))
b'\x89PNG\r\n\x1a\n'
```

Битовые операторы

Python предоставляет целочисленные операторы на уровне битов, аналогичные тем, что имеются в языке C. В табл. 12.6 собраны все операторы вместе с примерами для целочисленных переменных `a` (в десятичной системе счисления `5`, в двоичной — `0b0101`) и `b` (в десятичной системе счисления `1`, в двоичной — `0b0001`).

Таблица 12.6. 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 8. Оператор `&` возвращает биты, одинаковые в обоих аргументах, а оператор `|` возвращает биты, которые установлены в обоих аргументах. Оператор `^` возвращает биты, установленные в одном или в другом аргументе, но не в них обоих. Оператор `~` инвертирует порядок байтов в одном аргументе и 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.

Аналогия с ювелирными изделиями

Строки Unicode похожи на браслет с брелоками, а байты — на нитки с бусинами.

Читайте далее

Далее следует еще одна практическая глава — в ней показывается, как работать с датой и временем.

Упражнения

12.1. Создайте строку Unicode с именем `mystery` и присвойте ей значение `'\U0001f4a9'`. Выведите на экран значение строки `mystery`. Выведите на экран значение переменной `mystery` и ее имя Unicode.

12.2. Закодируйте строку `mystery`, на этот раз с использованием кодировки UTF-8, в переменную типа `bytes` с именем `pop_bytes`. Выведите на экран значение переменной `pop_bytes`.

12.3. 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`?

12.4. При работе с текстом могут пригодиться регулярные выражения. В следующем примере (пример 12.1) мы воспользуемся ими несколькими способами. Перед вами стихотворение Ode on the Mammoth Cheese, написанное Джеймсом Макинтайром в 1866 году во славу головки сыра весом 7000 фунтов, которая была изготовлена в Онтарио и отправлена в международное путешествие. Если не хотите самостоятельно перепечатывать это стихотворение, используйте свой любимый поисковик и скопируйте текст в программу. Или скопируйте стихотворение из проекта «Гутенберг» (http://bit.ly/mcintyre-poetry). Назовите следующую строку `mammoth`:

Пример 12.1. mammoth.txt

```
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.
```

12.5. Импортируйте модуль `re`, чтобы использовать функции регулярных выражений в Python. Примените функцию `re.findall()` для вывода на экран всех слов, начинающихся с буквы `c`.

12.6. Найдите все четырехбуквенные слова, которые начинаются с буквы `c`.

12.7. Найдите все слова, которые заканчиваются на букву `r`.

12.8. Найдите все слова, которые содержат три гласные подряд.

12.9. Используйте метод `unhexlify` для того, чтобы преобразовать шестнадцатеричную строку, созданную путем объединения двух строк для размещения на странице, в переменную типа `bytes` с именем `gif`:

```
'47494638396101000100800000000000ffffff21f9' +
'0401000000002c000000000100010000020144003b'
```

12.10. 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`. Корректен ли этот файл?

12.11. Ширина GIF-файла в пикселах является шестнадцатибитным целым числом с обратным порядком байтов, которое начинается со смещения 6 байт. 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`?

ГЛАВА 13
Календари и часы

«Раз!» — ударили часы на колокольной башне,
Те, что шестьдесят минут назад
Пробили полночь.

Фредерик Б. Нидхэм. The Round of the Clock

Хоть я и снялась для календаря, я никогда
не приходила вовремя.

Мэрилин Монро

Программисты прилагают удивительное количество усилий, работая с датами и временем. Поговорим о проблемах, с которыми постоянно приходится сталкиваться, а затем рассмотрим лучшие приемы для их решения.

Даты можно представить по-разному — способов не просто много, а слишком много. Даже англоговорящие люди, использующие римский календарь, имеют множество вариантов для написания простой даты:

- July 21 1987;
- 21 Jul 1987;
- 21/7/1987;
- 7/21/1987.

Наряду с другими проблемами, представление даты может быть двусмысленным. В приведенных примерах довольно легко понять, что 7 означает месяц, а 21 — день месяца, хотя бы потому, что у месяца не может быть номера 21. Но что насчет даты 1/6/2012? Мы говорим о 6 января или о 1 июня?

Название месяца в римском календаре изменяется в зависимости от языка. Даже год и месяц могут иметь разные определения в разных культурах.

Время также способно доставить неприятности, особенно из-за часовых поясов и переходов на летнее время. Если вы взглянете на карту часовых поясов, то окажется, что эти пояса больше соответствуют политическим и историческим границам, а не сменяются каждые 15° (360°/24) долготы. Кроме того, разные страны переходят на летнее время и обратно в разные дни года. Страны Южного полушария переводят свои часы вперед, когда страны Северного полушария переводят их назад, и наоборот (если вы немного подумаете, 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, что иногда приводит к путанице.

Високосный год

Високосные годы — еще одна проблема. Наверняка вы знаете, что каждый четвертый год — високосный (в такие годы проходят летние Олимпийские игры и выборы президента в США). Но знаете ли вы, что каждый сотый год не является високосным, а каждый 400-й — является? Рассмотрим пример кода, в котором проверяется, високосный год или нет:

```
>>> 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
```

Справка для любопытных:

- год состоит из 365,242 196 дня (после одного оборота вокруг Солнца Земля на четверть оборота вокруг своей оси отстает от места, где начинала движение);
- каждые четыре года добавляется один день. Теперь в среднем каждый год состоит из 365,242 196 – 0,25 = 364,992 196 дня;
- каждые 100 лет один день вычитается. Теперь средний год состоит из 364,992 196 + + 0,01 = 365,002 196 дня;
- каждые 400 лет добавляется один день. Теперь средний год состоит из 365,002 196 – – 0,0025 = 364,999 696 дня.

Получилось достаточно точно! О високосных секундах мы говорить не будем (https://alexwlchan.net/2019/05/falsehoods-programmers-believe-about-unix-time/).

Модуль datetime

Модуль стандартной библиотеки `datetime` позволяет работать с датами и временем (что вполне ожидаемо). В нем определены четыре основных класса объектов, и в каждом содержится 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(2019, 10, 31)
>>> halloween
datetime.date(2019, 10, 31)
>>> halloween.day
31
>>> halloween.month
10
>>> halloween.year
2019
```

Вы можете вывести на экран содержимое объекта `date` с помощью его метода `isoformat()`:

```
>>> halloween.isoformat()
'2019-10-31'
```

`iso` в данном контексте ссылается на ISO 8601 — международный стандарт для представления даты и времени. Согласно этому стандарту мы начинаем записывать дату с самого общего элемента (год), а заканчиваем самым точным (день). Вследствие чего есть возможность корректно отсортировать даты: сначала по году, затем по месяцу и 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 используется метод `today()` для генерации сегодняшней даты:

```
>>> from datetime import date
>>> now = date.today()
>>> now
datetime.date(2019, 4, 5)
```

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(2019, 4, 6)
>>> now + 17*one_day
datetime.date(2019, 4, 22)
>>> yesterday = now – one_day
>>> yesterday
datetime.date(2019, 4, 4)
```

Объект `date` может иметь значение в диапазоне, начинающемся с `date.min (year=1, month=1, day=1)` и заканчивающемся `date.max (year=9999, month=12, day=31)`. Таким образом, этот метод неприменим 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 (микросекунды). Если вы передадите не все аргументы, объект `time` предположит, что остальные имеют значение `0`. Кстати, то, что вы можете сохранять и получать микросекунды, не означает, что вы можете извлекать время из вашего компьютера с точностью до микросекунды. 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 напрямую, как показано в примере: 2 января, 2014, 3:04 утра, плюс 5 секунд и 6 микросекунд:

```
>>> from datetime import datetime
>>> some_day = datetime(2019, 1, 2, 3, 4, 5, 6)
>>> some_day
datetime.datetime(2019, 1, 2, 3, 4, 5, 6)
```

Объект `datetime` также имеет метод `isoformat()`:

```
>>> some_day.isoformat()
'2019-01-02T03:04:05.000006'
```

Буква `T`, которая находится в середине, разделяет дату и время.

Объект `datetime` имеет метод `now()`, с помощью которого можно получить текущие дату и время:

```
>>> from datetime import datetime
>>> now = datetime.now()
>>> now
```

```
datetime.datetime(2019, 4, 5, 19, 53, 7, 580562)
>>> now.year
2019
>>> now.month
4
>>> now.day
5
>>> now.hour
19
>>> now.minute
53
>>> now.second
7
>>> now.microsecond
580562
```

Объединить объекты `date` и `time` в объект `datetime` можно 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(2019, 4, 5, 12, 0)
```

Получить объекты `date` и `time` из объекта `datetime` можно с помощью методов `date()` и `time()`:

```
>>> noon_today.date()
datetime.date(2019, 4, 5)
>>> noon_today.time()
datetime.time(12,    0)
```

Модуль time

В Python есть модуль `datetime` с объектом `time`, но есть и отдельный модуль `time`. Более того, в модуле `time` имеется функция с каким бы вы думали именем? Правильно, `time()`.

Одним из способов представления абсолютного времени является подсчет количества секунд, прошедших с определенной начальной точки. В Unix считают *количество секунд*, прошедших с полуночи 1 января 1970 года[1]: это значение часто называют *epoch*. Как правило, данный способ самый простой для обмена датой и временем между системами.

Функция `time()` модуля `time` возвращает текущее время как значение `epoch`:

```
>>> import time
>>> now = time.time()
>>> now
1554512132.778233
```

[1] Примерно в это время появилась система Unix, если не обращать внимания на досадные високосные секунды.

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)
'Fri Apr  5 19:55:32 2019'
```

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, например с JavaScript. Однако иногда бывает нужно получить значения именно дней, часов и т. д., которые объект `time` предоставляет 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=2019, tm_mon=4, tm_mday=5, tm_hour=19,
tm_min=55, tm_sec=32, tm_wday=4, tm_yday=95, tm_isdst=1)
>>> time.gmtime(now)
time.struct_time(tm_year=2019, tm_mon=4, tm_mday=6, tm_hour=0,
tm_min=55, tm_sec=32, tm_wday=5, tm_yday=96, tm_isdst=0)
```

В моем (Центральном) часовом поясе 19:55 — это то же самое, что 00:55 следующего дня в поясе UTC (раньше его называли *Гринвичским временем* или *временем Зулу*). Если вы опустите аргумент в функциях `localtime()` или `gmtime()`, они предположат, что сконвертировать нужно текущее время.

Некоторые значения `tm_...` из структуры `struct_time` выглядят неоднозначно, поэтому обратитесь к табл. 13.1 для получения более подробной информации.

Таблица 13.1. Значения структуры struct_time

Индекс	Имя	Значение	Диапазон
0	tm_year	Год	от 0000 до 9999
1	tm_mon	Месяц	от 1 до 12
2	tm_mday	День месяца	от 1 до 31
3	tm_hour	Часы	от 0 до 23
4	tm_min	Минуты	от 0 до 59
5	tm_sec	Секунды	от 0 до 61
6	tm_wday	День недели	от 0 (понедельник) до 6 (воскресенье)
7	tm_yday	День года	от 1 до 366
8	tm_isdst	Летнее время?	0 = нет, 1 = да, –1 = неизвестно

Если вы не хотите вводить все эти имена, начинающиеся с конструкции `tm_`, структура `struct_time` также может играть роль именованного кортежа (см. раз-

дел «Именованные кортежи» на с. 219). Поэтому можно использовать индексы из предыдущей таблицы:

```
>>> import time
>>> now = time.localtime()
>>> now
time.struct_time(tm_year=2019, tm_mon=6, tm_mday=23, tm_hour=12,
tm_min=12, tm_sec=24, tm_wday=6, tm_yday=174, tm_isdst=1)
>>> now[0]
2019
print(list(now[x] for x in range(9)))
[2019, 6, 23, 12, 12, 24, 6, 174, 1]
```

Функция `mktime()` идет в другом направлении и преобразует объект `struct_time` в секунды `epoch`:

```
>>> tm = time.localtime(now)
>>> time.mktime(tm)
1554512132.0
```

Результат не совсем похож на предыдущее значение `epoch`, полученное с помощью функции `now()`, поскольку объект `struct_time` сохраняет время только 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 в UTC, не привязываясь к местному времени.

Еще один совет: *никогда не используйте летнее время*, если можно без этого обойтись. Если вы учитываете подобные переходы, в какой-то момент час у вас пропадет (весной), а в другой — наступит дважды (осенью). По разным причинам многие организации пользуются летним временем в своих компьютерных системах, а потом удивляются удвоению и потере данных из-за этого таинственного часа.

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)
'Fri Apr  5 19:58:23 2019'
```

Вы также можете преобразовывать дату и время с помощью функции `strftime()`. Она предоставляется как метод в объектах `datetime`, `date` и `time` и как функция в модуле `time`. Функция `strftime()` использует для вывода информации на экран спецификаторы формата, которые вы можете увидеть в табл. 13.2.

Таблица 13.2. 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

Слева к числам добавляется ноль.

Рассмотрим пример работы функции `strftime()`, предоставленной модулем `time`. Она преобразует объект `struct_time` в строку. Сначала мы определим строку формата `fmt`, и потом снова будем ее использовать:

```
>>> import time
>>> fmt = "It's %A, %B %d, %Y, local time %I:%M:%S%p"
>>> t = time.localtime()
>>> t
time.struct_time(tm_year=2019, tm_mon=3, tm_mday=13, tm_hour=15,
tm_min=23, tm_sec=46, tm_wday=2, tm_yday=72, tm_isdst=1)
>>> time.strftime(fmt, t)
"It's Wednesday, March 13, 2019, local time 03:23:46PM"
```

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(2019, 7, 4)
>>> fmt = "It's %B %d, %Y, local time %I:%M:%S%p"
>>> some_day.strftime(fmt)
"It's Friday, July 04, 2019, 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 в дату или время, 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`*, например `2019-01-29`.

Что произойдет, если строка даты, которую вы хотите проанализировать, имеет пробелы вместо дефисов?

```
>>> import time
>>> fmt = "%Y-%m-%d"
>>> time.strptime("2019 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 '2019 01 29' does not match format '%Y-%m-%d'
```

Будет ли довольна функция `strptime()`, если мы передадим ей несколько дефисов?

```
>>> time.strptime("2019-01-29", fmt)
time.struct_time(tm_year=2019, 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, но одно из значений находится вне диапазона, будет сгенерировано исключение:

```
>>> time.strptime("2019-13-29", 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 '2019-13-29' does not match format '%Y-%m-%d'
```

Имена соответствуют вашей *локали* — региональному набору настроек операционной системы. Чтобы вывести на экран другие названия месяцев и дней, измените свою локаль с помощью функции `setlocale()`: ее первый аргумент — `locale.LC_TIME` для даты и времени, а второй аргумент — это строка, содержащая сокращенное обозначение языка и страны. Давайте пригласим на вечеринку в честь Дня Всех Святых наших иностранных друзей. 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'
'Thursday, October 31'
'fr_fr'
'Jeudi, octobre 31'
'de_de'
'Donnerstag, Oktober 31'
'es_es'
'jueves, octubre 31'
'is_is'
'fimmtudagur, október 31'
>>>
```

Откуда можно взять эти волшебные значения аргумента `lang_country`? Следующий способ не самый надежный, но вы можете попробовать получить их все сразу (все несколько сотен):

```
>>> import locale
>>> names = locale.locale_alias.keys()
```

Из переменной `names` получим только имена локалей, которые, похоже, будут работать с методом `setlocale()`: например, те, которые мы использовали в предыдущем примере, — двухсимвольный код языка (http://bit.ly/iso-639-1), за которым следуют подчеркивание и двухсимвольный код страны (https://ru.wikipedia.org/wiki/ISO_3166-1_alpha-2):

```
>>> 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']
```

Если вы запустите функцию set_locale() и получите ошибку locale.Error: unsupported locale, это будет означать, что установка данной локали не поддерживается вашей операционной системой. Вам потребуется выяснить, что нужно операционной системе для добавления локали. Ошибка может произойти даже в том случае, если Python с помощью функции locale.locale_alias.keys() сообщит вам, что эта локаль хорошая. Я получил ошибку при тестировании на macOS с использованием локали cy_gb (валлийский, Великобритания), даже несмотря на то, что локаль is_is (исландский, Исландия) перед этим была принята.

Все преобразования

На рис. 13.1, взятом из вики Python (https://oreil.ly/C_39k), обобщены все стандартные преобразования времени, которые можно выполнить в Python.

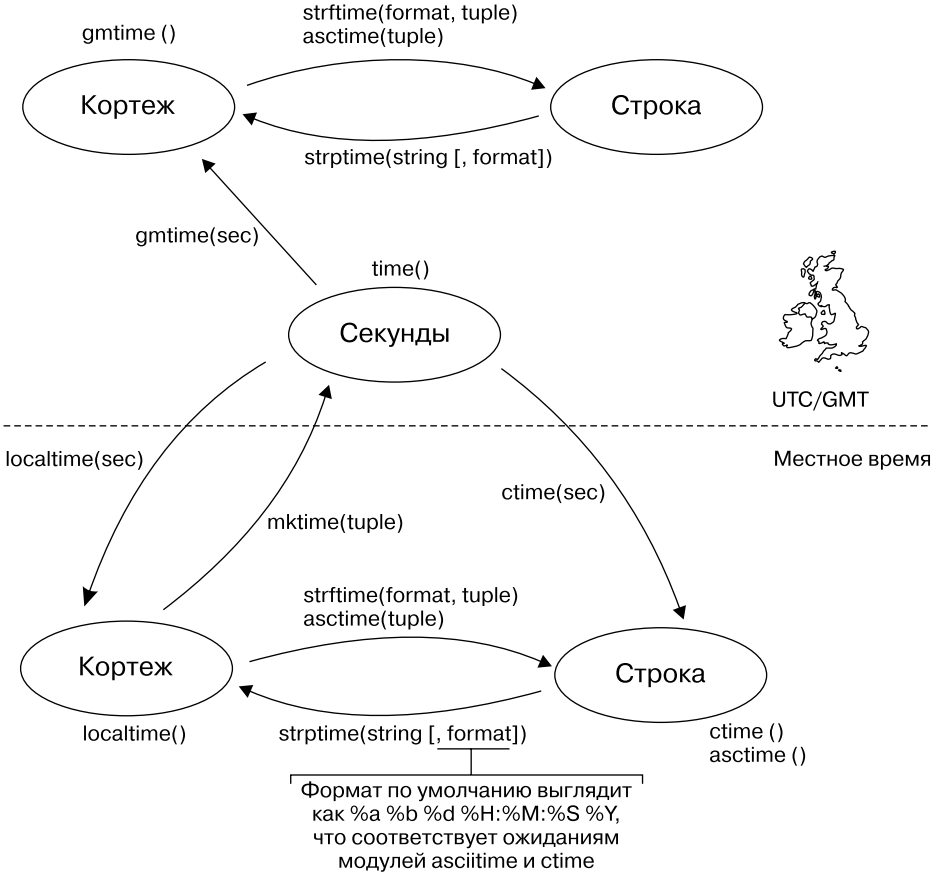

Рис. 13.1. Преобразования даты и времени

Альтернативные модули

Если вы считаете, что модули стандартной библиотеки только создают путаницу или в них не хватает какого-то конкретного преобразования, можете использовать альтернативные модули от сторонних разработчиков. Рассмотрим некоторые из них:

- `arrow` (https://arrow.readthedocs.io/). 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;
- `maya` (https://github.com/kennethreitz/maya). Интуитивный интерфейс для дат, времени и интервалов;
- `dateinfer` (https://github.com/jeffreystarr/dateinfer). Определяет правильные строки формата на основе строк, содержащих дату и время.

Читайте далее

Файлы и каталоги тоже хотят внимания.

Упражнения

13.1. Запишите текущие дату и время как строку в текстовый файл `today.txt`.

13.2. Прочтите текстовый файл `today.txt` и разместите данные в строке `today_string`.

13.3. Проанализируйте дату из строки `today_string`.

13.4. Создайте объект `date` с датой вашего рождения.

13.5. В какой день недели вы родились?

13.6. Когда вам будет (или уже было) 10 000 дней от роду?

ГЛАВА 14
Файлы и каталоги

> У меня есть файлы, у меня есть компьютерные файлы, есть, как вы понимаете, и файлы на бумаге. Но большая часть [истории] хранится у меня в голове. Помоги мне, Господь, если что-то случится с моей головой.
>
> *Джордж Р. Р. Мартин*

Когда вы только начинаете программировать, некоторые слова вы слышите раз за разом, но не уверены, имеют ли они какое-то техническое значение или просто сотрясают воздух. Термины *«файл»* и *«каталог»* входят в список таких слов, и они на самом деле имеют техническое значение. *Файл* — это последовательность байтов, хранящаяся в определенной *файловой системе*, к которой можно получить доступ по *имени файла*. *Каталог* — это коллекция файлов и, возможно, других каталогов. Слово *«папка»* является синонимом каталога: оно вошло в обиход, когда компьютеры получили графический пользовательский интерфейс, а чтобы вещи казались более знакомыми, использовались некоторые офисные концепции.

Многие файловые системы иерархичны и зачастую похожи на деревья. В реальных офисах, как правило, не бывает деревьев, и аналогия с папками работает только в том случае, если вы визуализируете все подкаталоги.

Ввод информации в файлы и ее вывод из них

Самый простой вид хранения — это старый добрый файл, иногда называемый *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. Как и во многих других языках, операции с файлами, присутствующие в этом языке программирования, в большинстве своем были смоделированы на основе знакомых и популярных аналогов, имеющихся в Unix.

Создаем или открываем файлы с помощью функции open()

Вам нужно вызвать функцию `open`, прежде чем сделать следующее:

- прочитать существующий файл;
- записать в новый файл;
- добавить данные в существующий файл;
- перезаписать существующий файл.

```
файл_об = open(имя_файла, режим)
```

Кратко поясню фрагменты этого вызова:

- *`файл_об`* — это объект файла, возвращаемый функцией `open()`;
- *`имя_файла`* — это строка, представляющая собой имя файла;
- *`режим`* — это строка, указывающая на тип файла и действия, которые вы хотите произвести над файлом.

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, чтобы гарантировать, что все операции записи завершены, а память освобождена. Далее вы узнаете, как оператор `with` может это автоматизировать.

В данной программе открывается файл с именем `oops.txt`, а затем закрывается без выполнения каких-либо действий. В результате будет создан пустой файл:

```
>>> fout = open('oops.txt', 'wt')
>>> fout.close()
```

Записываем в текстовый файл с помощью функции print()

Создадим заново файл `oops.txt`, но теперь запишем в него строку, а затем закроем:

```
>>> fout = open('oops.txt', 'wt')
>>> print('Oops, I created a file.', file=fout)
>>> fout.close()
```

Мы создали пустой файл `oops.txt` в предыдущем подразделе, поэтому программа перезаписала его. В вызове функции `print` мы использовали аргумент `file`. Без него функция `print` будет записывать данные в стандартный поток вывода, которым является ваша консоль (если только вы не указали программе оболочки перенаправить вывод данных в файл с помощью оператора `>` или не направили его в другую программу с помощью оператора `|`).

Записываем в текстовый файл с помощью функции write()

Мы использовали функцию `print` для того, чтобы записать строку в файл. Но можно использовать и функцию `write`.

В качестве примера многострочного источника данных возьмем лимерик о специальной теории относительности[1]:

```
>>> 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, в отличие от функции `print()`. Как и раньше, вы можете записать в текстовый файл несколько строк:

```
>>> fout = open('relativity', 'wt')
>>> print(poem, file=fout)
>>> fout.close()
```

Отсюда возникает вопрос: какую функцию использовать — `write()` или `print()`? Как вы уже видели, 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'`).

[1] В первой рукописи этой книги я использовал термин «общая теория относительности» и был любезно исправлен редактором-физиком.

Вместо значений по умолчанию мы используем пустые строки:

```
>>> fout = open('relativity', 'wt')
>>> print(poem, file=fout, sep='', end='')
>>> fout.close()
```

Если исходная строка большая, вы можете записывать в файл ее фрагменты (используя разделения (слайсы)) до тех пор, пока не запишете 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 первым заходом и оставшиеся 50 — вторым. Слайсы позволяют вам забраться за границы последовательности без генерации исключения.

Если файл `relativity` нам очень дорог, проверим, спасет ли режим `x` его от 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, используя функции read(), readline() и readlines()

Вы можете вызвать функцию `read()` без аргументов, чтобы достать весь файл сразу, 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. Мы будем считывать по 100 символов за раз и присоединять каждый фрагмент к строке `poem`, чтобы восстановить 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 считаете весь файл, дальнейшие вызовы функции `read()` будут возвращать пустую строку `' '`, которая рассматривается как `False` при проверке `if not fragment`. Это позволит выйти из цикла `while True`.

Вы также можете считывать файл построчно с помощью функции `readline()`. В следующем примере мы будем присоединять по одной строке к `poem`, чтобы восстановить 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 откроется в бинарном режиме. В этом случае вы будете читать и записывать байты, а не строки.

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 при работе 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 достаточно 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 файл, его закроет Python после того, как будет удалена последняя ссылка на этот файл. Таким образом, 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 таких объектов, как открытые файлы. 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 данном случае это одна строка), завершится (или нормально, или путем генерации исключения), файл закроется автоматически.

Меняем позицию с помощью функции seek()

В процессе чтения и записи Python отслеживает ваше местоположение в файле. Функция `tell()` возвращает ваше текущее смещение от начала файла в байтах. Функция `seek()` позволяет перейти к другому смещению в файле. Это значит, что

вам не обязательно прочитывать каждый байт файла, чтобы добраться до последнего, — вы можете использовать функцию `seek()`: сместиться именно к нужному 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), вы сместитесь на `offset` байтов от начала файла;
- если значение `origin` равно `1`, вы сместитесь на `offset` байтов с текущей позиции;
- если значение `origin` равно `2`, вы сместитесь на `offset` байтов от 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 последний байт можно считать 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 состоит не только из символов формата ASCII (каждый из которых занимает по 1 байту в памяти), вам будет трудно определить смещение. Ведь в таком случае оно будет зависеть от кодировки текста, а самая популярная кодировка (UTF-8) использует разное количество байтов для разных символов.

Отображение в памяти

Альтернативой чтению и записи файла является его *отображение в памяти (memory-map)* с помощью стандартного модуля `mmap`. Такой подход позволяет представить содержимое файла как `bytearray` в памяти. Для получения более подробной

информации обратитесь к документации (https://docs.python.org/3.7/library/mmap.html) и примерам (https://pymotw.com/3/mmap/index.html).

Операции с файлами

В Python, как и во многих других языках программирования, операции для работы с файлами основаны на аналогичных операциях ОС Unix. Некоторые функции, такие как `chown()` и `chmod()`, называются одинаково, но теперь появились и новые названия. Сначала я покажу, как Python решает задачи с помощью функций из модуля `os.path`, а затем — с помощью более нового модуля `pathlib`.

Проверяем существование файла с помощью функции exists()

Если хотите убедиться в том, что файл или каталог действительно существуют, а не являются плодом вашего воображения, воспользуйтесь функцией `exists()`. Передайте функции 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 подразделе, 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 *pathname (именем пути)* — полным именем файла, начинающимся с символа `/` и включающим в себя имена всех вложенных файлов. 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, используя функцию 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 функции link() или symlink()

В операционных системах семейства Unix файл находится в одном определенном месте, но может иметь несколько имен, называемых *ссылками*. В низкоуровневых *жестких ссылках* найти все имена заданного файла не так уж легко. *Символьная ссылка* — это альтернативный метод, который сохраняет новое имя в своем собственном файле, позволяя 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()` вносит изменения в права на доступ к файлу. Существуют права на чтение, запись и выполнение файла для пользователя (обычно того, кто создавал файл), а также для какой-то группы, в которой состоит пользователь, и для всех остальных. Команда принимает сильно сжатое восьмеричное значение (в системе счисления с основанием 8), в котором указаны пользователь, группа и другие, кто имеет доступ. Например, для указания того, что файл `oops.txt` для чтения доступен только своему владельцу, нужно ввести следующий код:

```
>>> os.chmod('oops.txt', 0o400)
```

Если вы не хотите работать с загадочными восьмеричными значениями и предпочитаете иметь дело с более понятными символами, можете импортировать некоторые константы из модуля `stat` и использовать такое выражение:

```
>>> 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)
```

Удаляем файл с помощью функции remove()

В этом фрагменте 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 такой иерархией и предоставляет функции, с помощью которых ею можно 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, используя функцию rmdir()

Немного подумав[1], вы решили, что этот каталог вам не нужен. Удалить его можно 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 (их 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, а просто убедитесь, что начинаете и заканчиваете соответствующими кавычками, одинарными или тройными):

```
>>> fout = open('poems/mcintyre/the_good_man', 'wt')
>>> fout.write('''Cheerful and happy was his mood,
```

[1] И почему же мы не подумали раньше?

```
... 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, проверьте, что получилось. Хорошо, если бы файл там был:

```
>>> 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, используя функцию glob()

Функция `glob()` ищет совпадающие имена файлов или каталогов, используя правила оболочки Unix, а не более полный синтаксис регулярных выражений. Эти правила выглядят так:

- `*` — совпадает со всем (в регулярных выражениях аналогом выступает `.*`);
- `?` — 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']
```

А как насчет файлов и каталогов с именами, состоящими из двух символов?

```
>>> glob.glob('??')
[]
```

Я думаю о слове из восьми букв, которое начинается на `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']
```

Pathname

Практически во всех компьютерах используется иерархическая файловая система, в которой каталоги (папки) содержат файлы и другие каталоги произвольного уровня вложенности. Если вы хотите обратиться к определенному файлу или каталогу, вам необходимо указать его путь (pathname): последовательность каталогов, необходимых для того, чтобы добраться до искомого объекта. Последовательность может быть *абсолютной* и начинаться сверху (от *корня*), а может быть *относительной* и начинаться с вашего текущего каталога.

Вы часто слышите, как люди путают *прямой слеш* (это символ `'/'`, а не музыкант из группы Guns N' Roses) и *обратный слеш* `'\'`[1]. В ОС семейств Unix и Macs (а также в URL) как *разделитель пути* применяется слеш, а в ОС Windows — обратный слеш[2].

Python позволяет использовать слеш в качестве разделителя пути, когда вы указываете имена. В Windows вы можете задействовать обратный слеш, но при этом обратный слеш также является распространенным символом в escape-последовательностях в Python, поэтому вы должны его удваивать или использовать необработанные строки Python:

```
>>> win_file = 'eek\\urk\\snort.txt'
>>> win_file2 = r'eek\urk\snort.txt'
>>> win_file
'eek\\urk\\snort.txt'
>>> win_file2
'eek\\urk\\snort.txt'
```

При создании pathname вы можете сделать следующее:

- использовать подходящий символ разделителя пути (`'/'` или `'\'`);
- построить путь (см. подраздел «Построение пути с помощью os.path.join()» далее в этой главе);
- использовать модуль `pathlib` (см. подраздел «Модуль pathlib» также в этой главе).

---

[1] Один из способов запомнить названия слешей: обычный наклонен вперед, а обратный — назад.

[2] QDOS — это операционная система, которую приобрел Билл Гейтс за $50 000, чтобы создать MS-DOS, когда IBM начала интересоваться первым ПК. Она была похожа на CP/M, где слеши использовались в аргументах командной строки. Когда в MS-DOS появились каталоги, для них пришлось использовать обратные слеши.

Получаем путь с помощью функции abspath()

Эта функция расширяет относительное имя до абсолютного. Если ваш текущий каталог — `/usr/gaberlunzie` и файл `oops.txt` находится там же, 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'
```

Построение пути с помощью функции os.path.join()

Когда вы создаете составное имя пути, вы можете вызвать функцию `os.path.join()` и объединить части попарно, используя правильный для вашей ОС символ-разделитель:

```
>>> import os
>>> win_file = os.path.join("eek", "urk")
>>> win_file = os.path.join(win_file, "snort.txt")
```

Если я запущу эту функцию на ПК с ОС Mac или Linux, я получу следующее:

```
>>> win_file
'eek/urk/snort.txt'
```

Запуск в ОС Windows даст такой результат:

```
>>> win_file
'eek\\urk\\snort.txt'
```

Но если один и тот же код выдает разные результаты в зависимости от того, где он был запущен, это может привести к проблемам. Новый модуль `pathlib` является портативным решением этой проблемы.

Модуль pathlib

В Python модуль `pathlib` добавился в версии 3.4. Он представляет собой альтернативу модулям `os.path`, которые мы только что рассмотрели. Зачем же нужен еще один модуль?

Вместо того чтобы считать пути строками, этот модуль создает объект типа `Path`, чтобы работать с ними на более высоком уровне. Создайте объект типа `Path` с помощью вызова `Path()`, а затем склейте все элементы вашего пути с помощью простых слешей (а не символов `'/'`):

```
>>> from pathlib import Path
>>> file_path = Path('eek') / 'urk' / 'snort.txt'
>>> file_path
PosixPath('eek/urk/snort.txt')
>>> print(file_path)
eek/urk/snort.txt
```

Этот прием со слешами использует «магические методы» Python. Объект класса `Path` может немного рассказать о себе:

```
>>> file_path.name
'snort.txt'
>>> file_path.suffix
'.txt'
>>> file_path.stem
'snort'
```

Вы можете передать переменную `file_path` в функцию `open()`, как и любое имя файла или строку пути.

Вы также можете увидеть, что случится, если запустить эту программу в другой операционной системе или если вам нужно сгенерировать внешние пути на своем компьютере:

```
>>> from pathlib import PureWindowsPath
>>> PureWindowsPath(file_path)
PureWindowsPath('eek/urk/snort.txt')
>>> print(PureWindowsPath(file_path))
eek\urk\snort.txt
```

Для получения более подробной информации обратитесь к документации (https://docs.python.org/3/library/pathlib.html).

BytesIO и StringIO

Вы уже знаете, как изменять данные в памяти и как получать данные из файлов, а также записывать их внутрь. Но что делать в ситуации, когда у вас есть данные в памяти, но вам нужно вызвать функцию, которая ожидает в качестве параметра файл (или наоборот)? Вам понадобится изменить данные и передать эти байты или символы, не создавая временных файлов.

Вы можете применить `io.BytesIO` для бинарных данных (байтов) и `io.StringIO` для текстовых данных (строк). И в том и в другом случае из данных будет создан *файлоподобный объект*, который подойдет для всех функций работы с файлами, рассмотренных в этой главе.

Одним из вариантов использования этих типов является преобразование формата данных. Поработаем, например, с библиотекой PIL (более подробно о ней вы сможете прочитать в подразделе «PIL и Pillow» на с. 475), которая считывает и записывает данные из изображения.

В качестве первого аргумента методов `open()` и `save()` класса `Image` ожидается имя файла или *файлоподобный объект*. Код в примере 14.1 использует тип `BytesIO` для того, чтобы считать и записать данные, лежащие в памяти. Он считывает один или несколько файлов изображений из командной строки, преобразовывает данные из них в три разных формата и выводит длины и первые 10 байт полученного результата.

Пример 14.1. convert_image.py

```
from io import BytesIO
from PIL import Image
import sys

def data_to_img(data):
    """Return PIL Image object, with data from in-memory <data>"""
    fp = BytesIO(data)
    return Image.open(fp)    # выполняет чтение из памяти

def img_to_data(img, fmt=None):
    """Return image data from PIL Image <img>, in <fmt> format"""
    fp = BytesIO()
    if not fmt:
        fmt = img.format     # сохраняет оригинальный формат
    img.save(fp, fmt)        # записывает в память
    return fp.getvalue()

def convert_image(data, fmt=None):
    """Convert image <data> to PIL <fmt> image data"""
    img = data_to_img(data)
    return img_to_data(img, fmt)

def get_file_data(name):
    """Return PIL Image object for image file <name>"""
    img = Image.open(name)
    print("img", img, img.format)
    return img_to_data(img)

if __name__ == "__main__":
    for name in sys.argv[1:]:
        data = get_file_data(name)
        print("in", len(data), data[:10])
        for fmt in ("gif", "png", "jpeg"):
            out_data = convert_image(data, fmt)
            print("out", len(out_data), out_data[:10])
```

Поскольку объект BytesIO ведет себя как файл, вы можете вызывать для него функции seek(), read() и write() так же, как и для обычного файла. Если вы вызовете функцию seek(), а затем функцию read(), то получите только байты, которые находятся между текущей позицией файла и его концом. Функция getvalue() вернет все байты объекта BytesIO.

Перед вами полученные значения для изображения, которое вы увидите в главе 20:

```
$ python convert_image.py ch20_critter.png
img <PIL.PngImagePlugin.PngImageFile image mode=RGB size=154x141
    at 0x10340CF28> PNG
in 24941 b'\\x89PNG\\r\\n\\x1a\\n\\x00\\x00'
out 14751 b'GIF87a\\x9a\\x00\\x8d\\x00'
out 24941 b'\\x89PNG\\r\\n\\x1a\\n\\x00\\x00'
out 5914 b'\\xff\xd8\\xff\\xe0\\x00\\x10JFIF'
```

Читайте далее

Следующая глава будет чуточку сложнее. В ней мы рассмотрим конкурентность (способы выполнения нескольких операций одновременно) и процессы (запущенные программы).

Упражнения

14.1. Выведите на экран список всех файлов из текущего каталога.

14.2. Выведите на экран список всех файлов из родительского каталога.

14.3. Присвойте строку `This is a test of the emergency text system` переменной `test1` и запишите эту переменную в файл с именем `test.txt`.

14.4. Откройте файл `test.txt` и считайте его содержимое в строку `test2`. Будут ли одинаковыми строки `test1` и `test2`?

ГЛАВА 15

Данные во времени: процессы и конкурентность

> Что может компьютер, но не может большинство людей, — лежать на складе, запечатанным в коробке.
>
> *Джек Хэнди*

Эта глава, а также две последующие — более сложные, чем остальные. Сейчас мы рассмотрим данные во времени (последовательный и конкурентный доступ на одном компьютере), в главе 16 поговорим о данных в коробке (хранение и извлечение данных с помощью особых файлов и баз данных), а в главе 17 — о данных в пространстве (сети).

Программы и процессы

Когда вы запускаете отдельную программу, ваша операционная система создает один *процесс*, который использует системные ресурсы (центральный процессор, память, место на диске) и структуры данных в *ядре* операционной системы (файлы и сетевые соединения, статистику 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, выделяя каждому из них немного времени и 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, такого как Mac's Activity Monitor в OS X, диспетчера задач в Windows или команды `top` в Linux.

Вы также можете получить доступ к данным процесса ваших собственных программ. Модуль стандартной библиотеки `os` предоставляет общий способ доступа к определенной системной информации. Например, следующие функции получают *идентификатор процесса* и *текущую рабочую папку* запущенного интерпретатора Python:

```
>>> import os
>>> os.getpid()
76051
>>> os.getcwd()
'/Users/williamlubanovic'
```

А это — 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. Запускать и останавливать другие существующие в Python программы можно, используя модуль `subprocess` из стандартной библиотеки. Если вы хотите просто запустить другую программу в оболочке и получить результат ее работы (как стандартный отчет, так 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 вы хотите запустить еще какую-нибудь программу или процесс, которые займут много времени, обратитесь к разделу «Конкурентность» на с. 308. Поскольку аргументом функции `getoutput()` является строка, представляющая собой команду оболочки, вы можете включить в эту строку 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 не строку, а объект типа `bytes` и не использует оболочку:

```
>>> ret = subprocess.check_output(['date', '-u'])
>>> ret
b'Mon Mar 31 04:01:50 UTC 2014\n'
```

Чтобы показать статус завершения другой программы, используйте функцию `getstatusoutput()`, которая возвращает кортеж с кодом статуса и результатом 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`.

Запускать программы с аргументами можно двумя способами. Первый заключается в том, чтобы разместить аргументы 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 `shell=True`, чтобы распознать команду `date -u`, разбив ее на отдельные строки и, возможно, развернув любые символы подстановки, такие как * (в нашем примере мы их не использовали).

Второй метод создает список аргументов, поэтому нет необходимости вызывать оболочку:

```
>>> ret = subprocess.call(['date', '-u'])
Tue Jan 21 04:41:59 UTC 2014
```

Создаем процесс с помощью модуля multiprocessing

Запустить функцию Python как отдельный процесс или даже создать несколько независимых процессов можно с помощью модуля `multiprocessing`. Рассмотрим короткий и простой пример 15.1. Сохраните его под именем `mp.py`, а затем запустите, введя команду `python mp.py`:

Пример 15.1. mp.py

```
import multiprocessing
import os

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()` вызвала новый процесс и запустила в нем функцию `do_this()`. Поскольку мы делали это в цикле с четырьмя итерациями, сгенерировалось четыре новых процесса, которые выполнили `do_this()` и завершились.

У модуля `multiprocessing` есть много возможностей. Он прекрасно подходит для тех случаев, когда в целях экономии времени нужно распределить какие-то задачи между несколькими процессами: это может быть, например, загрузка веб-страницы, изменение размера изображений или что-то другое. В модуле содержатся способы постановки задач в очередь, включения взаимодействия между процессами и ожидания завершения всех процессов.

В разделе «Конкурентность» на с. 308 содержится более подробная информация.

Убиваем процесс, используя функцию terminate()

Если вы создали один или несколько процессов, а теперь по какой-то причине хотите их завершить (возможно, они зависли в цикле или же вам стало скучно, а может, просто захотелось побыть жестоким правителем), используйте функцию `terminate()`. В примере 15.2 процесс должен досчитать до миллиона, замирая после каждого шага на секунду и выводя раздражающее сообщение. Однако у нашей основной программы заканчивается терпение, и она сбивает его с орбиты:

Пример 15.2. mp2.py

```
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!
```

Получаем системную информацию с помощью модуля os

Стандартный пакет `os` предоставляет подробную информацию о вашей системе и позволяет управлять некоторыми функциями, запуская скрипт, написанный на Python, от лица привилегированного пользователя (например, администратора). Помимо функций файлов и каталогов, которые мы рассмотрели в главе 14, в нем есть такие информационные функции (они запускаются на iMac):

```
>>> import os
>>> os.uname()
posix.uname_result(sysname='Darwin',
nodename='iMac.local',
release='18.5.0',
version='Darwin Kernel Version 18.5.0: Mon Mar 11 20:40:32 PDT 2019;
  root:xnu-4903.251.3~3/RELEASE_X86_64',
machine='x86_64')
>>> os.getloadavg()
(1.794921875, 1.93115234375, 2.2587890625)
>>> os.cpu_count()
4
```

Имеется также полезная функция `system()`, которая выполняет командную строку так, как если бы вы ввели ее в консоли:

```
>>> import os
>>> os.system('date -u')
Tue Apr 30 13:10:09 UTC 2019
0
```

В этом пакете много полезного. Обратитесь к документации (https://docs.python.org/3/library/os.html), чтобы узнать все самое интересное.

Получаем информацию о процессах с помощью модуля psutil

Сторонний пакет psutil (https://github.com/giampaolo/psutil) также предоставляет информацию о системе и процессах для ОС Linux, Unix, macOS, и Windows.

Нетрудно догадаться, как он устанавливается:

```
$ pip install psutil
```

В нем содержатся следующие разделы:

- *Система*. ЦП, память, диск, сеть, сенсоры.
- *Процессы*. Идентификатор, родительский идентификатор, ЦП, память, открытые файлы, потоки.

Мы уже видели (в разделе, посвященном пакету `os`), что мой компьютер имеет четыре процессора. Сколько времени (в секундах) они использовали?

```
>>> import psutil
>>> psutil.cpu_times(True)
[scputimes(user=62306.49, nice=0.0, system=19872.71, idle=256097.64),
scputimes(user=19928.3, nice=0.0, system=6934.29, idle=311407.28),
scputimes(user=57311.41, nice=0.0, system=15472.99, idle=265485.56),
scputimes(user=14399.49, nice=0.0, system=4848.84, idle=319017.87)]
```

Насколько они заняты сейчас?

```
>>> import psutil
>>> psutil.cpu_percent(True)
26.1
>>> psutil.cpu_percent(percpu=True)
[39.7, 16.2, 50.5, 6.0]
```

Вам, возможно, никогда не понадобятся подобные данные, но всегда полезно знать, как их можно получить.

Автоматизация команд

Вы часто выполняете команды из оболочки (вводя их вручную или запуская скрипты оболочки). Однако в Python имеется несколько качественных сторонних инструментов для управления запуском. Связанная с этим тема — *очереди задач* — рассматривается в подразделе «Очереди» на с. 309.

Invoke

Первая версия инструмента `fabric` позволяет вам определить локальные и удаленные (сетевые) задачи в коде Python. Разработчик разбил этот пакет на `fabric2` (для удаленных задач) и `invoke` (для локальных задач).

Вы можете установить `invoke` с помощью следующей команды:

```
$ pip install invoke
```

Одним из вариантов использования пакета `invoke` является возможность сделать функции доступными в качестве аргументов командной строки. Давайте создадим файл `tasks.py`, который содержит строки, показанные в примере 15.3.

Пример 15.3. tasks.py

```
from invoke import task

@task
```

```
def mytime(ctx):
    import time
    now = time.time()
    time_str = time.asctime(time.localtime(now))
    print("Local time is", timestr)
```

(Аргумент `ctx` — первый аргумент для каждой преобразуемой функции, но используется он только самим пакетом `invoke`. Неважно, с каким именем, но этот аргумент там должен быть.)

```
$ invoke mytime
Local time is Thu May  2 13:16:23 2019
```

Используйте аргументы `-l` или `-list`, чтобы увидеть, какие задачи доступны:

```
$ invoke -l
Available tasks:

  mytime
```

Задачи могут иметь аргументы, и вы можете вызвать из командной строки несколько задач одновременно (аналогично использованию оператора && в командной строке).

Другие варианты использования:

- запуск локальных скриптов оболочки с помощью функции `run`;
- ответ на возвращаемые другими программами строковые шаблоны.

Этот пакет мы рассмотрели поверхностно. Для получения более подробной информации обратитесь к документации (http://docs.pyinvoke.org/en/stable/).

Другие вспомогательные методы для команд

Следующие пакеты немного похожи на `invoke`, но в определенных ситуациях лучшее решение может предложить один из таких пакетов:

- click (https://click.palletsprojects.com/en/7.x/);
- doit (https://pydoit.org/);
- sh (http://amoffat.github.io/sh/);
- delegator (https://github.com/amitt001/delegator.py);
- pypeln (https://cgarciae.github.io/pypeln/).

Конкурентность

Официальный сайт Python рассматривает тему конкурентности в целом и с точки зрения стандартной библиотеки (http://bit.ly/concur-lib). На страницах сайта содержится множество ссылок на различные пакеты и приемы, а я в этой главе остановлюсь на самых полезных пакетах.

Когда речь идет о компьютерах, находиться в ожидании приходится по одной из двух причин:

- из-за *ограничения ввода-вывода*. Это наиболее распространенная причина. Процессоры компьютеров безумно быстры — в сотни раз быстрее, чем компьютерная память, и в тысячи — чем диски или сети;
- из-за *ограничения процессора*. Это случается при выполнении таких *объемных* задач, как научные или графические расчеты.

С конкурентностью связаны еще два термина:

- *синхронность* — одна вещь следует за другой, как гусята, семенящие за родителями;
- *асинхронность* — задачи независимы, как гуси, которые плавают в пруду.

По мере того как от простых систем и задач вы будете переходить к проблемам реальной жизни, вам все чаще придется иметь дело с конкурентностью. Возьмем, например, сайт. Обычно вы можете довольно быстро предоставлять веб-клиентам статическую и динамическую страницы. Если ожидание длится долю секунды, приложение считается интерактивным, однако 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 хотя бы немного медленнее обычного.

Ну а если вы не можете повлиять на то, что долго выполняется? Например, на загрузку файла на сервер, на изменение размеров изображения или запрос к базе данных? Вы больше не можете делать это при помощи синхронного кода вашего веб-сервера, поскольку кто-то уже ждет.

Если вам нужно выполнить несколько задач как можно быстрее на одном компьютере, вы можете сделать их независимыми, и тогда медленные задачи не будут блокировать остальные.

Ранее в этой главе было показано, как использовать многопроцессорную обработку для параллельной работы на одной машине. Если нужно, например, изменить размер изображения, ваш веб-сервер может создать отдельный процесс для данной задачи 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. Особый подход нужен для обработки ошибок, поскольку конкурентные вычисления сложнее обычных. Многое может пойти не так, и ваши шансы на успех будут ниже.

Какие же методы могут помочь справиться с этими сложностями? Начнем с хорошего способа для управления несколькими задачами — это *очереди*.

Очереди

Очередь похожа на список: элементы добавляются с одного ее конца и удаляются с другого. Часто такой принцип называют *FIFO* (first in, first out — «первым пришел, первым ушел»).

Представьте, что вы моете посуду. Если вы хотите совершить полный цикл, вам нужно вымыть тарелку, высушить ее и отложить в сторону. Сделать это можно несколькими способами. Можно вымыть, высушить и отложить первую тарелку и повторить эти действия со второй и последующими тарелками. Или же можно *сгруппировать* операции: сначала вымыть всю посуду, затем ее высушить и отложить в сторону. При этом подразумевается, что в раковине и сушилке достаточно места, чтобы разместить там всю посуду на каждом этапе. Оба подхода являются синхронными — один работник выполняет одно действие в любой момент времени.

В качестве альтернативы вы могли бы найти одного-двух помощников. Предположим, вы моете тарелку, передаете ее тому, кто тарелки сушит, а сушильщик передаст тарелку тому, кто отложит ее в сторону. Если все работают в одном темпе, вы должны закончить работу гораздо быстрее, чем если бы выполняли ее полностью самостоятельно.

Но что, если вы моете посуду быстрее, чем сушильщик успевает с ней справляться? Тогда или вымытая посуда будет падать на пол, или вы будете складывать ее между собой и сушильщиком. Есть вариант, что вы просто начнете что-нибудь насвистывать до тех пор, пока сушильщик не примет от вас очередную тарелку. А если последний помощник работает медленнее сушильщика, падать на пол или накапливаться будет уже сухая посуда либо насвистывать начнет уже сушильщик. Хоть у вас и 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. Увеличение количества работников поможет быстрее построить сарай или вымыть посуду. При этом будут задействованы *очереди*.

В целом, очереди переносят *сообщения*, которые могут содержать любую информацию. В данном случае нас интересуют очереди для распределенного управления задачами, также известные как *рабочие очереди* или *очереди заданий*. Каждая тарелка из раковины выдается доступному мойщику: он ее моет и передает первому доступному сушильщику. Тот, в свою очередь, отдает тарелку первому доступному работнику, в чьи обязанности входит убрать ее в сторону. Этот процесс может быть синхронным (работники сначала ждут, когда им дадут тарелку, а потом 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

Реализовать очередь можно разными способами. Для одного компьютера модуль стандартной библиотеки `multiprocessing` (с которым вы ранее уже встречались) содержит функцию `Queue`. Смоделируем ситуацию только с одним мойщиком посуды и несколькими сушильщиками (кто-то позже отложит посуду в сторону) и создадим промежуточную очередь `dish_queue`. Назовем эту программу `dishes.py` (пример 15.4).

Пример 15.4. 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()
```

Запускаем нашу 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, производящий наборы тарелок. В действительности здесь создаются отдельные процессы, которые общаются между собой. Я использовал `JoinableQueue` и финальный метод `join()`, чтобы сообщить мойщику о том, что вся посуда высушена. У модуля `multiprocessing` есть очереди и других типов: узнать о них вы можете из документации (https://docs.python.org/3/library/multiprocessing.html).

Потоки

Поток запускается внутри процесса и имеет доступ ко всему, что находится в процессе, — это можно сравнить с раздвоением 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 применения потоков (пример 15.5).

Пример 15.5. thread1.py

```
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 воспроизвести пример с посудой, основанный на процессах, используя потоки (пример 15.6).

Пример 15.6. thread_dishes.py

```
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 таких языках, как C и C++, они могут вызвать появление ошибок, которые ужасно трудно найти и исправить. Для того чтобы задействовать потоки, весь код программы — и код внешних библиотек, которые он использует, — должен быть *потокобезопасным*. В предыдущем примере кода потоки не имели дела с глобальными переменными, поэтому 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 существовании другого, и в любой момент каждое привидение может просматривать, добавлять, удалять или перемещать любую вещь из домашней обстановки.

Вы настороженно идете по дому, снимая показания своими впечатляющими инструментами. И внезапно замечаете, что подсвечник, мимо которого вы проходили несколько секунд назад, пропал.

Содержимое дома похоже на переменные в программе. Привидения — это потоки процесса (дома). Если бы привидения только просматривали содержимое дома, это не было бы проблемой — как поток, который просто читает 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 могут так изменить вещи в других комнатах, что вы даже не заметите.

Несмотря на все ваши причудливые инструменты, вам будет очень трудно разобраться в том, кто, когда и как это сделал.

Если бы вместо потоков вы использовали несколько процессов, это можно было бы сравнить с несколькими домами, в каждом из которых обитает только одно (живое) существо. Если поставить бутылку бренди перед камином, через час она все еще будет там. Возможно, немного жидкости испарится, но сама бутылка останется на том же месте.

Потоки могут быть полезны и безопасны, когда речь не идет о глобальных данных. В частности, рекомендуется использовать потоки для экономии времени при ожидании завершения некоторых операций ввода/вывода. В этом случае потокам не нужно сражаться за данные, поскольку у каждого из них имеется свой набор переменных.

Но у потоков иногда есть веские причины для изменения глобальных данных. Фактически самая распространенная причина для использования нескольких потоков — это возможность разделить между ними работу над определенными данными, поэтому ожидается, что какие-то 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. Нужно лишь не забыть ее разблокировать. Блокировки также могут быть вложенными — что, если и другая бригада охотников за привидениями будет наблюдать за этой же комнатой или за всем домом? Использование блокировок достаточно традиционно, но 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.

Итак, для Python рекомендации следующие:

- используйте потоки для задач, связанных с ограничениями ввода-вывода;
- используйте процессы, сетевые вычисления или события (которые мы рассмотрим в следующем подразделе) для задач, связанных с ограничениями процессора.

Concurrent.futures

Как вы только что видели, использование потоков или нескольких процессов требует отслеживания большого количества деталей. Модуль `concurrent.futures` был добавлен в стандартную библиотеку в версии Python 3.2 для того, чтобы упростить эту задачу. Он позволяет вам спроектировать асинхронный пул так называемых работников с помощью потоков (с ограничением по вводу-выводу) и процессов (с ограничением по ЦП). Вы получаете объект типа `future`, который позволяет отслеживать состояние заданий и собирать результаты работы.

В примере 15.7 вы видите тестовую программу, которую можно сохранить под именем `cf.py`. Функция-задача `calc()` спит в течение одной секунды (наш способ симулировать бурную деятельность), вычисляет квадратный корень своего аргумента и возвращает его. Программа принимает необязательный аргумент командной строки — количество работников, которых следует задействовать (по умолчанию `-3`), запускает заданное количество из пула потоков, затем из пула процессов, а затем выводит на экран затраченное время. Список `values` содержит пять чисел, отправленных функции `calc()` по одному с помощью работника-потока или работника-процесса.

Пример 15.7. cf.py

```
from concurrent import futures
import math
import time
import sys

def calc(val):
    time.sleep(1)
```

```
    result = math.sqrt(float(val))
    return result

def use_threads(num, values):
    t1 = time.time()
    with futures.ThreadPoolExecutor(num) as tex:
        results = tex.map(calc, values)
    t2 = time.time()
    return t2 - t1

def use_processes(num, values):
    t1 = time.time()
    with futures.ProcessPoolExecutor(num) as pex:
        results = pex.map(calc, values)
    t2 = time.time()
    return t2 - t1

def main(workers, values):
    print(f"Using {workers} workers for {len(values)} values")
    t_sec = use_threads(workers, values)
    print(f"Threads took {t_sec:.4f} seconds")
    p_sec = use_processes(workers, values)
    print(f"Processes took {p_sec:.4f} seconds")

if __name__ == '__main__':
    workers = int(sys.argv[1])
    values = list(range(1, 6)) # 1 .. 5
    main(workers, values)
```

Я получил следующие результаты:

```
$ python cf.py 1
Using 1 workers for 5 values
Threads took 5.0736 seconds
Processes took 5.5395 seconds
$ python cf.py 3
Using 3 workers for 5 values
Threads took 2.0040 seconds
Processes took 2.0351 seconds
$ python cf.py 5
Using 5 workers for 5 values
Threads took 1.0052 seconds
Processes took 1.0444 seconds
```

Вызов функции `sleep()` заставил каждого работника потратить секунду на каждое вычисление.

- Если мы задействуем всего одного работника, вычисления выполнятся последовательно, а общее время составит более пяти секунд.
- Если мы задействуем пять работников, то их количество совпадет с количеством проверяемых значений, поэтому программа будет выполняться чуть дольше чем одну секунду.

- Если мы задействуем трех работников, нам понадобятся два запуска, чтобы обработать все пять значений, поэтому пройдет всего две секунды.

В программе я проигнорировал реальные результаты (рассчитанные нами квадратные корни) для того, чтобы сделать акцент на затраченном времени. Важно отметить, что использование функции `map()` для определения пула заставит нас дожидаться выполнения действий всех работников перед тем, как возвращать результаты. Если вы хотите получать каждый отдельный результат по завершении работы, создайте еще одну тестовую программу (назовем ее `cf2.py`), в которой каждый работник будет возвращать значения по мере их вычисления (пример 15.8).

Пример 15.8. cf2.py

```
from concurrent import futures
import math
import sys

def calc(val):
    result = math.sqrt(float(val))
    return val, result

def use_threads(num, values):
    with futures.ThreadPoolExecutor(num) as tex:
        tasks = [tex.submit(calc, value) for value in values]
        for f in futures.as_completed(tasks):
             yield f.result()

def use_processes(num, values):
    with futures.ProcessPoolExecutor(num) as pex:
        tasks = [pex.submit(calc, value) for value in values]
        for f in futures.as_completed(tasks):
             yield f.result()

def main(workers, values):
    print(f"Using {workers} workers for {len(values)} values")
    print("Using threads:")
    for val, result in use_threads(workers, values):
        print(f'{val} {result:.4f}')
    print("Using processes:")
    for val, result in use_processes(workers, values):
        print(f'{val} {result:.4f}')

if __name__ == '__main__':
    workers = 3
    if len(sys.argv) > 1:
        workers = int(sys.argv[1])
    values = list(range(1, 6)) # 1 .. 5
    main(workers, values)
```

Наши функции `use_threads()` и `use_processes()` теперь являются функциями-генераторами, которые вызывают `yield` для того, чтобы вернуть значение на каждой итерации. По результатам одного запуска на моей машине вы можете увидеть, что работники не всегда выполняют задачи в порядке от 1 до 5:

```
$ python cf2.py 5
Using 5 workers for 5 values
Using threads:
3 1.7321
1 1.0000
2 1.4142
4 2.0000
5 2.2361
Using processes:
1 1.0000
2 1.4142
3 1.7321
4 2.0000
5 2.2361
```

Вы можете использовать модуль `concurrent.futures` всякий раз, когда вам нужно запустить несколько конкурирующих задач, например таких, как:

- поиск (crawling) URL в Интернете;
- обработка файлов, например изменение размера изображений;
- вызов API какой-либо службы.

Как обычно, в документации (https://docs.python.org/3/library/concurrent.futures.html) вы можете найти дополнительную информацию с большим количеством технических деталей.

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 является программирование, *основанное на событиях* (event-based programming). Программа, основанная на событиях, запускает центральный *цикл обработки событий*, раздает задачи и повторяет цикл. Так устроен веб-сервер NGINX, работающий быстрее, чем Apache.

Библиотека `gevent` основана на событиях и позволяет достичь следующего: вы пишете обычный императивный код, и волшебным образом его части превращаются в *сопрограммы*. Они похожи на генераторы, которые могут взаимодействовать друг с другом и отслеживать свое текущее состояние. Библиотека `gevent` модифицирует многие стандартные объекты Python, такие как `socket`, для того чтобы использовать их механизм вместо блокирования. Это не работает для кода надстроек Python, который написан на C, например для некоторых драйверов баз данных.

Вы можете установить библиотеку `gevent` с помощью `pip`:

```
$ pip install gevent
```

Вот вариант примера 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` (пример 15.9).

Пример 15.9. 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 запустить их можно асинхронно благодаря версии функции `gethostbyname()` из библиотеки `gevent`.

Запустите файл `gevent_test.py`:

```
$ python gevent_test.py
66.6.44.4
74.125.142.121
78.136.12.50
```

Функция `gevent.spawn()` создает *гринлет (greenlet)* (называемый также *зеленым потоком* и *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 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 версии `socket` из модуля `gevent` вы можете использовать его функции, называемые *monkey-patching (обезьяний патч)*. Они модифицируют стандартные модули, такие как `socket`, для использования гринлетов вместо того, чтобы каждый раз вызывалась версия 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()
```

Таким образом, все обычные сокеты заменятся на сокеты gevent даже в стандартной библиотеке. Опять же это работает только для кода Python, но не для библиотек, написанных на C.

Еще одна функция monkey-patching даже больше 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 (пример 15.10).

Пример 15.10. 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:

```
$ python 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, такие как Pinterest, используют gevent для значительного ускорения своей работы. Используйте gevent строго по назначению, как таблетки по рецепту.

Чтобы увидеть больше примеров, обратитесь к этому руководству — http://sdiehl.github.io/gevent-tutorial/.

Существует 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 без применения традиционных веб-серверов, таких как Apache.

twisted

`twisted` (http://twistedmatrix.com/trac/) — это асинхронный фреймворк, управляемый событиями, для работы с сетями. Вы подключаете функции к таким событиям, как получение данных или закрытие соединения, и функции вызываются, когда событие происходит. В данном случае речь идет о *функциях обратного вызова*. Если вы уже писали код на языке JavaScript, это может показаться вам знакомым. По мере роста приложения некоторым разработчикам бывает сложнее управлять кодом, основанным на функциях обратного вызова.

Чтобы установить фреймворк, введите следующую команду:

```
$ pip 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 обратимся к серверу `knock_server.py` (пример 15.11).

Пример 15.11. knock_server.py

```
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` (пример 15.12).

Пример 15.12. 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()
```

Первым запустим сервер:

```
$ python knock_server.py
```

Вторым — клиент:

```
$ python 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 `twisted`, попробуйте запустить другие примеры из его документации.

asyncio

Библиотека `asyncio` была добавлена в версию Python 3.4. Она представляет собой способ определения конкурентного кода с помощью новой функциональности, предоставляемой ключевыми словами `async` и `await`. Это очень обширная тема с большим количеством деталей. Чтобы не перегружать эту главу, я переместил рассмотрение библиотеки `asyncio` и связанных с ней тем в приложение B.

Redis

Приведенные ранее примеры кода для мытья посуды, где использовались процессы или потоки, запускались на одной машине. Рассмотрим еще один подход к очередям, которые могут запускаться на одной машине или во всей сети. Иногда даже для нескольких процессов и/или потоков одной машины бывает недостаточно. Вы можете воспринимать этот подраздел как мостик между конкурентным использованием одного блока (компьютера) и конкурентным использованием нескольких блоков.

Для запуска примеров из этого раздела вам понадобятся сервер Redis и его модуль для Python. Чтобы узнать, как их скачать, обратитесь к подразделу «Redis» на с. 256. В этой главе Redis используется как база данных. Здесь же мы рассмотрим его возможности для работы с конкурентностью.

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» (blocking pop)*. Если список пуст, то все они просто тратят время впустую. Как только сообщение появляется, его получает первый ожидающий работник.

Как и в предыдущих, основанных на процессах и потоках примерах, код файла `redis_washer.py` генерирует последовательность посуды (пример 15.13).

Пример 15.13. redis_washer.py

```
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 с названиями тарелок, за которыми следует заключительное сообщение со словом `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` (пример 15.14).

Пример 15.14. 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 сообщения, в котором первым токеном будет слово `dishes`, и выводит сообщение о каждой высушенной тарелке. Подчиняясь сообщению `quit`, он завершает цикл.

Запустите сначала сушильщика, а затем мойщика. Использование символа `&` в конце команды ставит первую программу в *фоновый режим*: она продолжает выполняться, 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* (строки `quit`) является числом. После того как процесс сушильщика считывает этот идентификатор `quit`, он завершает работу и выводит на терминал дополнительную информацию о фоновом процессе (что также зависит от системы). Вы можете использовать контрольное значение (или по-другому — некорректное значение), чтобы указать на что-то особенное в потоке данных. В нашей ситуации мы говорим, что закончили работу. В противном случае придется добавить больше программной логики, например:

- заранее оговорить некоторое максимальное количество посуды, что также будет похоже на контрольное значение;
- выполнять определенную 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 вместо того, чтобы ожидать контрольного значения.

Новый файл `redis_dryer2.py` показан в примере 15.15.

Пример 15.15. 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
```

Через 20 секунд другие процессы-сушильщики получают значение `None` от вызова `blpop` (это указывает на то, что они завершились по таймеру). Процессы выводят свои последние сообщения и завершаются:

```
Dryer process 44447 is done
Dryer process 44446 is done
```

После того как завершается последний подпроцесс-сушильщик, завершается и основная программа-сушильщик:

```
[1]+  Done                    python redis_dryer2.py
```

Помимо очередей

С увеличением числа работающих элементов растет вероятность сбоев в работе нашего конвейера. Хватит ли нам работников, если нужно будет вымыть посуду после банкета? А что, если сушильщики напьются до чертиков? А если забьется раковина? Ох уж эти проблемы!

Как же с ними справиться? К счастью, есть некоторые приемы.

- *Запустить и забыть*. Просто передавайте обработанные объекты дальше и не заботьтесь о последствиях, даже если там 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 графиком, в противном случае вы услышите звук бьющейся посуды. Вы можете добавлять в список ожидания новые задачи, а какой-то процесс будет доставать из этого списка последнее сообщение и помещать его в список обработки. Обработанное сообщение будет удалено из списка обработки и добавлено в список завершенных задач. Это позволит вам узнать, какие задачи не были выполнены или затрачивают слишком много времени. Вы можете сделать это самостоятельно с помощью Redis или использовать систему, которую кто-то другой уже написал и протестировал. Некоторые основанные на Python пакеты для работы с очередями (часть из них используют Redis) позволяют удобно управлять процессом:

- celery (http://www.celeryproject.org/) может выполнять распределенные задачи как синхронно, так и асинхронно, используя рассмотренные нами методы — `multiprocessing`, `gevent` и др.;
- rq (http://python-rq.org/) — это библиотека Python для очередей задач, также основанная на Redis.

Читайте далее

В этой главе мы пропустили данные сквозь процессы. В следующей главе вы увидите, как сохранять и получать данные, используя разные форматы файлов и баз данных.

Упражнения

15.1. Используйте модуль `multiprocessing`, чтобы создать три отдельных процесса. Заставьте каждый из них подождать случайное количество секунд между нулем и единицей, вывести текущее время, а затем завершить работу.

ГЛАВА 16

Данные в коробке: надежные хранилища

> 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 дорогая и требующая постоянного питания: если питание пропадет, то все данные, которые в ней хранятся, будут утеряны. Жесткие диски медленнее оперативной памяти, но более емкие, стоят дешевле и могут хранить данные даже после того, как кто-то выдернет шнур питания. Поэтому много усилий при создании компьютерных систем было потрачено на поиск оптимального соотношения между хранением данных на диске и в оперативной памяти. Как программистам, нам важно *постоянство*: хранение и извлечение данных с использованием энергонезависимых носителей, таких как диски.

В этой главе мы рассмотрим разнообразные способы хранения данных, каждый из которых оптимизирован для разных целей: плоские файлы, структурированные файлы и базы данных. Операции с файлами, не касающиеся ввода-вывода, рассматриваются в главе 14.

Запись — это термин, обозначающий некоторые связанные между собой данные. Запись состоит из отдельных *полей*.

Плоские текстовые файлы

Самый простой пример постоянного хранилища — это старый добрый файл, который иногда называют еще плоским файлом. Он хорошо работает в том случае, когда у данных очень простая структура и вы полностью записываете их на диск или считываете с него. Такой подход годится для простых текстовых данных.

Текстовые файлы, дополненные пробелами

В этом формате каждое поле записи имеет фиксированную длину и при необходимости дополняется до требуемой длины (как правило, пробелами) так, чтобы все записи имели одинаковый размер. Программист может использовать функцию `seek()` для перемещения по файлу, для записи или только для чтения необходимых записей и полей.

Структурированные текстовые файлы

Для простых текстовых файлов единственным уровнем организации является строка. Но иногда вам может понадобиться более структурированный файл, чтобы сохранить данные из программы для дальнейшего использования или отправить их другой программе.

Существует множество форматов, и у каждого есть свои особенности.

- *Разделитель* (separator или delimiter) — такие символы, как табуляция (`'\t'`), запятая (`','`), вертикальная черточка (`'|'`). Это пример CSV — формата со значениями, разделенными запятой.
- Символы `'<'` и `'>'` в окружении *тегов*. Примеры включают в себя XML и HTML.
- Знаки препинания. Примером является JavaScript Object Notation (JSON).
- Выделение пробелами. Примером является YAML (аббревиатура расшифровывается как YAML Ain't Markup Language — «YAML — не язык разметки»).
- Другие 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, такую как список или словарь. Но лучшим решением будет использовать стандартный модуль `csv`, поскольку парсинг этих файлов может оказаться сложнее, чем вы думаете. Ознакомьтесь с важными характеристиками файлов CSV, о которых нужно помнить:

- некоторые имеют альтернативные разделители вместо запятой: самыми популярными являются `'|'` и `'\t'`;
- некоторые имеют escape-последовательности. Если символ-разделитель встречается внутри поля, все поле может быть окружено кавычками или же ему будет предшествовать escape-последовательность;

- некоторые имеют разные символы конца строк. В Unix используется `'\n'`, в Microsoft — `'\r \n'`. Apple раньше применяла символ `'\r'`, но теперь перешла на использование `'\n'`;
- некоторые в первой строке могут иметь названия столбцов.

Сначала мы посмотрим, как читать и записывать список строк, каждая из которых содержит список столбцов:

```
>>> 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']]
```

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 столбцы, разделенные запятыми, и ряды, разделенные символами перевода строки.

Данные могут иметь формат списка словарей, а не списка списков. Снова считаем файл `villains`, на этот раз используя новую функцию `DictReader()` и указывая имена столбцов:

```
>>> import csv
>>> with open('villains', 'rt') as fin:
...     cin = csv.DictReader(fin, fieldnames=['first', 'last'])
...     villains = [row for row in cin]
...
```

```
>>> print(villains)
[OrderedDict([('first', 'Doctor'), ('last', 'No')]),
OrderedDict([('first', 'Rosa'), ('last', 'Klebb')]),
OrderedDict([('first', 'Mister'), ('last', 'Big')]),
OrderedDict([('first', 'Auric'), ('last', 'Goldfinger')]),
OrderedDict([('first', 'Ernst'), ('last', 'Blofeld')])]
```

Словарь `OrderedDict` используется для обеспечения совместимости с версиями Python ниже 3.6, в которых словари сохраняли порядок элементов по умолчанию.

Перепишем CSV-файл с помощью новой функции `DictWriter()`. Мы также вызовем функцию `writeheader()`, чтобы записать начальную строку, содержащую имена столбцов, 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.csv` со строкой заголовка (пример 16.1).

Пример 16.1. villains.csv

```
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 столбцов и соответствующие ключи словаря:

```
>>> import csv
>>> with open('villains', 'rt') as fin:
...     cin = csv.DictReader(fin)
...     villains = [row for row in cin]
...
>>> print(villains)
[OrderedDict([('first', 'Doctor'), ('last', 'No')]),
OrderedDict([('first', 'Rosa'), ('last', 'Klebb')]),
OrderedDict([('first', 'Mister'), ('last', 'Big')]),
OrderedDict([('first', 'Auric'), ('last', 'Goldfinger')]),
OrderedDict([('first', 'Ernst'), ('last', 'Blofeld')])]
```

XML

Файлы с разделителями отображают только два измерения: ряды (строки) и столбцы (поля внутри строк). Если вы хотите обмениваться структурами данных между программами, вам нужен способ кодирования иерархий, последовательностей, множеств и других структур в виде 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 контент размещается после *начального тега*, такого как `<menu>`. Имеется и соответствующий *конечный тег*, такой как `</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 начального 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 `pancakes` для 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 — атрибутов, значений или тегов-потомков — 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*, у него есть такие подформаты, как RSS и Atom. В некоторых отраслях существует множество специализированных форматов XML, например в сфере в финансов (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 стандартный модуль `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 меню
3
>>> len(root[0])        # количество блюд для завтрака
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 Python для работы с XML можно отметить следующие:

- `xml.dom`. The Document Object Model (DOM) (знакомая разработчикам на JavaScript) представляет веб-документы в виде иерархических структур. Этот модуль загружает XML-файл в память целиком и позволяет получать доступ ко всем его частям;
- `xml.sax`. Simple API for XML, или SAX, разбирает XML на ходу, поэтому не загружает в память сразу весь документ. Она может быть хорошим выбором, если нужно обработать очень большие потоки XML.

Примечание о безопасности XML

Вы можете использовать любой формат, описанный в этой главе, чтобы сохранять объекты в файлы и снова их читать. Однако при этом существует вероятность получить проблемы с безопасностью.

Например, в следующем фрагменте XML-файла со страницы «Википедии» об атаках billion laughs (https://en.wikipedia.org/wiki/Billion_laughs_attack) (это разновидность атаки «отказ в обслуживании») определяется десять вложенных сущностей, каждая из которых расширяет более низкий уровень в 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: атака подорвет работоспособность всех XML-библиотек, упомянутых в предыдущем подразделе. На ресурсе Defused XML (https://bitbucket.org/tiran/defusedxml) эта и другие атаки перечислены наряду с уязвимостями библиотек Python. Перейдя по ссылке, вы увидите, как изменять настройки многих библиотек так, чтобы избежать подобных проблем. Вы также можете использовать библиотеку `defusedxml` в качестве внешнего интерфейса безопасности для других библиотек:

```
>>> # небезопасно:
>>> from xml.etree.ElementTree import parse
>>> et = parse(xmlfile)
>>> # безопасно:
>>> from defusedxml.ElementTree import parse
>>> et = parse(xmlfile)
```

Стандартный сайт Python также имеет свою собственную страницу об уязвимостях XML.

HTML

Огромные объемы данных сохраняются в формате гипертекстового языка разметки — Hypertext Markup Language (HTML). Это основной формат документов в Интернете. Проблема заключается в том, что значительная часть этих документов не соответствует правилам формата HTML, что затрудняет анализ. Кроме того, большая

часть HTML предназначена для форматирования выводимой информации, а не для обмена данными. Поскольку эта глава предназначена для описания относительно хорошо определенных форматов данных, я вынес рассмотрение HTML в главу 18.

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 прекрасно подходит Python, что делает его хорошим выбором для обмена данными между программами. Вы увидите множество примеров JSON для веб-разработки в главе 18.

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 более раннего примера 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.

Вы можете получить исключение, пытаясь закодировать или декодировать некоторые объекты, например `datetime` (этот вопрос детально рассматривается в главе 13), как показано здесь:

```
>>> import datetime
>>> import json
>>> now = datetime.datetime.utcnow()
>>> now
datetime.datetime(2013, 2, 22, 3, 49, 27, 483336)
>>> json.dumps(now)
Traceback (most recent call last):
# ... (опустили стек вызовов, чтобы спасти деревья)
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` в то, что JSON понимает, например в строку или значение времени `epoch` (см. главу 13):

```
>>> 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 обычно сконвертированными типами данных, может быть неудобно выполнять такие особые преобразования. Вы можете изменить способ кодирования JSON с помощью наследования, описанного в главе 10. Документация JSON для Python (https://docs.python.org/3.3/library/json.html) содержит пример такого переопределения для комплексных чисел, что также заставляет JSON притвориться мертвым. Напишем переопределение для `datetime`:

```
>>> import datetime
>>> now = datetime.datetime.utcnow()
>>> 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:

```
>>> import datetime
>>> now = datetime.datetime.utcnow()
>>> 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
```

При работе с JSON и другими форматами структурированного текста вы можете загрузить файл в память и разместить его в структуре данных, не зная о самих структурах заранее. Затем вы можете пройтись по структурам, используя функцию isinstance() и методы, соответствующие типу, чтобы проверить значения структур. Например, если один из элементов является словарем, вы можете извлечь его содержимое с помощью функций keys(), values() и items().

После того как вы сделали это сложным способом, сообщу вам, что существует более простой способ преобразовать объекты типа `datetime` в JSON:

```
>>> import datetime
>>> import json
>>> now = datetime.datetime.utcnow()
>>> json.dumps(now, default=str)
'"2019-04-17 21:54:43.617337"'
```

Инструкция `default=str` указывает функции `json.dumps()` применить функцию преобразования `str()` к тем типам данных, которые она не понимает. Это сработает, поскольку в определении класса `datetime.datetime` присутствует метод `__str__()`.

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 и 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.
```

Такие значения, как `true`, `false`, `on` и `off`, преобразуются в булевы переменные. Целые числа и строки преобразуются в их эквиваленты в Python. Для остального 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, а 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 Неда Батчелдера 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.

Tablib

Теперь, когда вы прочитали все предыдущие разделы, я расскажу вам, что существует сторонний пакет, который позволяет импортировать, экспортировать и изменять табличные данные в форматах CSV, JSON или YAML[1], а также данные в Microsoft Excel, Pandas DataFrame и некоторые другие. Вы можете установить его привычным способом (`pip install tablib`), а также заглянуть в документацию (http://docs.python-tablib.org/en/master/).

Pandas

Сейчас самое время познакомиться с Pandas (https://pandas.pydata.org/) — библиотекой Python для структурированных данных. Это отличный инструмент для решения реальных проблем с данными. Она позволяет:

- читать и записывать данные во множестве текстовых и бинарных форматов, таких как:
  - текст, поля которого разделены запятыми (CSV), символами табуляции (TSV) или другими символами;
  - текст фиксированной длины;
  - Excel;
  - JSON;
  - таблицы HTML;
  - SQL;
  - HDF5;
  - и др. (https://pandas.pydata.org/pandas-docs/stable/user_guide/io.html);

[1] Однако на данный момент XML не поддерживается.

- группировать, разбивать, объединять, разделять, сортировать, выбирать и помечать;
- преобразовывать типы данных;
- изменять размер или форму;
- обрабатывать случаи, когда данные отсутствуют;
- генерировать случайные значения;
- управлять временными последовательностями.

Функции чтения возвращают объект типа `DataFrame` (https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.html). Это является стандартным представлением для двумерных данных (которые делятся на строки и столбцы) в Pandas. Объект этого типа похож на электронную таблицу или таблицу реляционной базы данных. Его одномерный младший брат называется `Series` (https://pandas.pydata.org/pandas-docs/stable/reference/series.html).

В примере 16.2 показывается простое приложение, которое считывает данные из нашего файла `villains.csv` из примера 16.1.

Пример 16.2. Читаем данные в формате CSV с помощью Pandas

```
>>> import pandas
>>>
>>> data = pandas.read_csv('villains.csv')
>>> print(data)
    first        last
0  Doctor          No
1    Rosa       Klebb
2  Mister         Big
3   Auric  Goldfinger
4   Ernst     Blofeld
```

Переменная `data` имеет тип `DataFrame`: у этого типа данных возможностей больше, чем у простого словаря Python. Он особенно полезен для обработки большого количества чисел с помощью NumPy, а также для подготовки данных для машинного обучения.

Обратитесь к разделам *Getting Started* (https://pandas.pydata.org/pandas-docs/stable/getting_started/index.html) документации Pandas, чтобы узнать подробнее о ее особенностях, и к разделу *10 Minutes to Pandas* (https://pandas.pydata.org/pandas-docs/stable/getting_started/10min.html) для того, чтобы увидеть рабочие примеры.

Воспользуемся Pandas для того, чтобы создать небольшой календарь — список, содержащий первый день первых трех месяцев 2019 года:

```
>>> import pandas
>>> dates = pandas.date_range('2019-01-01', periods=3, freq='MS')
>>> dates
DatetimeIndex(['2019-01-01', '2019-02-01', '2019-03-01'],
  dtype='datetime64[ns]', freq='MS')
```

Создать такой календарь можно было бы и с помощью функций даты и времени, которые мы рассмотрели в главе 13. Но это намного сложнее, особенно отладка (дата и время добавляют работы). Pandas также позволяет обрабатывать множество особых деталей даты и времени (https://pandas.pydata.org/pandas-docs/stable/user_guide/timeseries.html), например бизнес-месяцы и годы.

Мы еще поговорим о Pandas, когда речь пойдет о картах (см. подраздел «Geopandas» на с. 498) и научных приложениях (см. раздел «Pandas» на с. 516).

Конфигурационные файлы

Большинство программ предлагают различные *параметры* или *настройки*. Динамические настройки могут быть переданы как аргументы программы, но долговременные настройки должны где-то храниться. Соблазн на скорую руку определить собственный формат *конфигурационного файла* очень силен, но вы должны устоять. Как правило, это бывает и неточно, и не так уж 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 подразделах.

Здесь мы используем стандартный модуль `configparser`, который обрабатывает файлы с расширением `.ini`, характерные для Windows. Такие файлы имеют разделы с определениями *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, в том числе более мощная интерполяция. Обратитесь к документации `configparser` (http://bit.ly/configparser). Если вам нужно более двух уровней вложенности, попробуйте использовать YAML или JSON.

Бинарные файлы

Некоторые файловые форматы были разработаны для хранения определенных структур данных и не являются ни реляционными базами данных, ни базами данных NoSQL. В следующих подразделах рассказывается о некоторых из них.

Заполненные пробелами бинарные файлы и управление памятью. Такие файлы похожи на заполненные пробелами текстовые файлы, но содержимое может быть бинарным, а в качестве заполнителя может использоваться байт `\x00`. Каждая запись имеет фиксированный размер, как и каждое поле внутри записи. Это позволяет легче искать нужные записи и поля с помощью функции `seek()`. Каждая операция с данными выполняется вручную, поэтому такой подход должен применяться только в очень низкоуровневых (близких к «железу») ситуациях.

Данные в таком формате могут быть размещены в ОЗУ с помощью стандартной библиотеки `mmap`. Взгляните на примеры (https://pymotw.com/3/mmap/) и стандартную документацию (https://docs.python.org/3.7/library/mmap.html).

Электронные таблицы

Электронные таблицы, в частности Microsoft Excel, — это широко распространенный бинарный формат данных. Если вы можете сохранить свою таблицу в CSV-файл, то можете считать его с помощью стандартного модуля `csv`, который был описан ранее.

Это распространяется на бинарный файл `xls`: для его считывания и записи можно использовать стороннюю библиотеку `xlrd` (https://pypi.org/project/xlrd/) или `tablib` (она упоминалась ранее в подразделе «Tablib» на с. 338).

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. Вам могут быть полезными следующие модули:

- `h5py` — интерфейс низкого уровня с широкими возможностями. Прочтите его документацию (http://www.h5py.org/) и код (https://github.com/h5py/h5py);

- `PyTables` — 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 главе 22 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 высчитывать крупные объемы данных. Хорошим примером использования этого формата является Million Song Dataset (http://bit.ly/millionsong) с записями песен в форматах HDF5 и SQLite.

TileDB

У формата HDF5 недавно появился последователь, который позволяет хранить как плотные, так и разреженные массивы — TileDB (https://tiledb.com/). Установите интерфейс Python (https://github.com/TileDB-Inc/TileDB-Py) (он включает в себя и саму библиотеку TileDB), запустив команду `pip install tiledb`. Эта библиотека предназначена для работы с научными данными и приложениями.

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 определены *схемами* и имеют *ограничения*.
- *Объединения* позволяют найти отношения между различными типами данных.
- Декларативный (в противоположность императивному) язык запросов SQL (Structured Query Language).

Такие базы данных называются *реляционными*, поскольку они показывают отношения между различными типами данных в форме прямоугольных *таблиц*. Например, в нашем более раннем примере меню есть отношение между каждым элементом и его ценой.

Таблица представляет собой прямоугольную сетку *столбцов* (полей данных) и *строк* (отдельных записей), похожую на электронную таблицу. Пересечение строки и столбца называется *ячейкой*. Чтобы создать таблицу, необходимо указать ее имя и порядок, имена и типы ее столбцов. Каждая строка имеет одинаковые столбцы, хотя столбец может быть определен так, чтобы в ячейках отсутствовали данные

(`null`). В примере с меню вы могли бы создать таблицу, содержащую по одной строке для каждого продаваемого элемента. Каждый элемент имеет одинаковые столбцы, включая и тот, который хранит цену.

Первичным ключом таблицы является столбец или группа столбцов. Значения ключа должны быть уникальными — таким образом предотвращается ввод одинаковых данных в таблицу. Этот ключ *индексируется* для быстрого поиска во время выполнения запроса. Работа индекса немного похожа на работу алфавитного указателя, который позволяет быстро найти определенный ряд.

Каждая таблица находится внутри родительской *базы данных*, как файлы в каталоге. Два уровня иерархии позволяют немного лучше организовывать данные.

Да, словосочетание «база данных» используется в разных значениях: называет и сервер, и хранилище таблиц, и сами данные. Если вам нужно говорить обо всех них одновременно, можно использовать термины «сервер базы данных», «база данных» и «данные».

Если вы хотите найти строки по определенному неключевому значению, определите для столбца *вторичный индекс*. В противном случае база данных должна будет выполнить *сканирование таблицы* — поиск нужного значения перебором всех строк.

Таблицы могут быть связаны друг с другом с помощью *внешних ключей*, и значения столбцов могут быть ограничены этими ключами.

SQL

SQL не является API или протоколом. Это декларативный *язык*: вы говорите, *что* вам нужно, а не *как* это сделать. SQL — универсальный язык реляционных баз данных. Запросы SQL являются текстовыми строками: клиент отсылает их серверу базы данных, а тот определяет, что с ними делать дальше.

Существует несколько стандартов определения SQL, но все поставщики баз данных добавили свои собственные настройки и расширения, что привело к появлению 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.

Есть две основные категории утверждений SQL.

- *DDL (Data Definition Language — язык определения данных)*. Обрабатывает создание, удаление, ограничения и разрешения для таблиц, баз данных и пользователей.
- *DML (Data Manipulation Language — язык манипулирования данными)*. Обрабатывает добавление данных, их выборку, обновление и удаление.

В табл. 16.1 перечислены основные команды SQL DDL.

Таблица 16.1. Основные команды SQL DDL

Операция	Шаблон SQL	Пример SQL
Создание базы данных	CREATE DATABASE *имя_базы*	CREATE DATABASE d
Выбор текущей базы данных	USE *имя_базы*	USE d
Удаление базы данных и ее таблиц	DROP DATABASE *имя_базы*	DROP DATABASE d
Создание таблицы	CREATE TABLE *имя_таблицы* (*описания_столбцов*)	CREATE TABLE t (id INT, count INT)
Удаление таблицы	DROP TABLE *имя_таблицы*	DROP TABLE t
Удаление всех строк таблицы	TRUNCATE TABLE *имя_таблицы*	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 их можно было отличить от имен столбцов.

Основные операции DML реляционной базы данных можно запомнить с помощью акронима CRUD:

- *C*reate — создание с помощью оператора SQL `INSERT`;
- *R*ead — чтение с помощью `SELECT`;
- *U*pdate — обновление с помощью `UPDATE`;
- *D*elete — удаление с помощью `DELETE`.

В табл. 16.2 показаны команды, доступные SQL DML.

Таблица 16.2. Основные команды SQL DML

Операция	Шаблон SQL	Пример SQL
Добавление строки	INSERT INTO *имя_таблицы* VALUES(...)	INSERT INTO t VALUES(7, 40)
Выборка всех строк и столбцов	SELECT * FROM *имя_таблицы* SELECT * FROM t	SELECT * FROM t
Выборка всех строк и некоторых столбцов	SELECT cols FROM *имя_таблицы*	SELECT id, count FROM t
Выборка некоторых строк и некоторых столбцов	SELECT cols FROM *имя_таблицы* WHERE *условие*	SELECT id, count from t WHERE count > 5 AND id = 9
Изменение некоторых строк в столбце	UPDATE *имя_таблицы* SET col = *значение* WHERE *условие*	UPDATE t SET count = 3 WHERE id = 5
Удаление некоторых строк	DELETE FROM *имя_таблицы* WHERE *условие*	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 такие аргументы, как имя пользователя, пароль, адреса сервера и пр.;
- `cursor()` — создание объекта курсора, предназначенного для работы с запросами;
- `execute()` и `executemany()` — запуск одной или нескольких команд SQL;
- `fetchone()`, `fetchmany()` и `fetchall()` — получение результатов работы функции `execute()`.

Модули базы данных в Python, которые будут рассмотрены в следующих подразделах, соответствуют DB-API, но часто имеют некоторые расширения или разницу в деталях.

SQLite

SQLite (http://www.sqlite.org/) — это хорошая легкая реляционная база данных с открытым исходным кодом. Она реализована как стандартная библиотека Python и хранит базы данных в обычных файлах. Эти файлы можно переносить в другие машины и операционные системы, что делает SQLite портативным решением для простых приложений 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 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 удобно 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 процветающим бизнесом по содержанию придорожного контактного зоопарка. В таблице будут следующие столбцы:

- `critter` — строка переменной длины, наш первичный ключ;
- `count` — целочисленное количество единиц используемого инвентаря для этого животного;

- `damages` — сумма, выраженная в долларах, наших убытков 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, таких как запросы 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>
```

На этот раз мы использовали в запросе три вопросительных знака, чтобы показать, что мы планируем вставить три значения, а затем передать эти значения в виде кортежа в функцию `execute()`. Заполнители помогают нам справляться с утомительными деталями, например с расстановкой кавычек. Они защищают от *внедрений SQL-кода* — внешней атаки, распространенной в Сети, которая внедряет в систему вредные команды SQL (такие атаки часто происходят в Интернете).

Теперь проверим, сможем ли мы вывести всех наших животных снова:

```
>>> 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 еще раз, но 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 бы подумали, что это медведи. Лучше всегда проверять фактические данные.

Перед тем как попрощаться с SQLite, следует очиститься. Если мы открывали соединение и курсор, их нужно закрыть после того, как работа будет закончена:

```
>>> curs.close()
>>> conn.close()
```

MySQL

MySQL (http://www.mysql.com/) — очень популярная реляционная база данных с открытым исходным кодом. В отличие от SQLite она является настоящим сервером, поэтому клиенты могут получать к ней доступ с разных устройств всей сети.

В табл. 16.3 перечислены драйверы, которые вы можете использовать для того, чтобы получить доступ к MySQL из Python. За более подробной информацией обо всех драйверах MySQL для Python обратитесь к энциклопедии на `python.org` (https://wiki.python.org/moin/MySQL).

Таблица 16.3. Драйверы MySQL

Название	Ссылка	Пакет PyPi	Импортировать как	Примечание
mysqlclient	https://mysqlclient.readthedocs.io	mysql-connector-python	MySQLdb	
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	Требует наличия клиентской библиотеки MySQL C

PostgreSQL

PostgreSQL (http://www.postgresql.org/) — полнофункциональная реляционная база данных с открытым исходным кодом, гораздо более продвинутая, чем MySQL. В табл. 16.4 показаны драйверы Python, которые можно использовать для того, чтобы получить к ней доступ.

Таблица 16.4. Драйверы PostgreSQL

Название	Ссылка	Пакет PyPi	Импортировать как	Примечание
psycopg2	http://initd.org/psycopg	psycopg2	psycopg2	Необходим pg_config из клиентских инструментов PostgreSQL
py-postgresq	https://pypi.org/project/py-postgresql	py-postgresq	py-postgresq	

Самым популярным драйвером является psycopg2, но для его установки требуется наличие клиентских библиотек PostgreSQL.

SQLAlchemy

SQL не для всех реляционных баз данных одинаков, а 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/).

Она не является стандартной, тем не менее широко известна и многими используется. Вы можете установить ее в свою систему с помощью следующей команды:

```
$ pip install sqlalchemy
```

Использовать SQLAlchemy можно на нескольких уровнях.

- Самый низкий уровень управляет пулами соединений с базами данных, выполняет команды SQL и возвращает результат. Это ближе всего к DB-API.
- Следующий уровень — *язык выражений SQL*, который позволяет вам выражать запросы более Python-ориентированным способом.
- Самый высокий уровень — это ORM (Object Relational Model — объектно-реляционная модель), 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 подразделах. Вам не нужно импортировать драйвер — он будет определен с помощью строки соединения, которую вы предоставите SQLAlchemy. Эта строка выглядит примерно так:

```
диалект + драйвер :// пользователь : пароль @ хост : порт / имя_базы
```

В нее нужно поместить следующие значения:

- *диалект* — тип базы данных;
- *драйвер* — драйвер, который вы хотите использовать для этой базы данных;

- *пользователь* и *пароль* — строки аутентификации для этой базы данных;
- *хост* и *порт* — расположение сервера базы данных (значение `port` нужно указывать только в том случае, если вы используете нестандартный порт);
- *имя_базы* — имя базы данных, к которой нужно подключиться.

В табл. 16.5 перечислены диалекты и драйверы.

Таблица 16.5. Соединение с SQLAlchemy

Диалект	Драйвер
sqlite	pysqlite (можно опустить)
mysql	Mysqlconnector
mysql	Pymysql
mysql	Oursql
postgresql	psycopg2
postgresql	Pypostgresql

Почитайте более подробно о диалектах SQLAlchemy для MySQL (https://docs.sqlalchemy.org/en/13/dialects/mysql.html), SQLite (https://docs.sqlalchemy.org/en/13/dialects/sqlite.html), PostgreSQL (https://docs.sqlalchemy.org/en/13/dialects/postgresql.html) и других баз данных (https://docs.sqlalchemy.org/en/13/dialects/).

Уровень движка. Сначала обратимся к самому низкому уровню 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 *хост*, *порт*, *имя_пользователя* и *пароль*. Имя *имя_базы* информирует SQLite о том, какой файл использовать для хранения вашей базы данных. Если вы опустите параметр *имя_базы*, SQLite создаст базу данных в памяти. Если *имя_базы* начинается со слеша `/`, значит, это абсолютное имя файла на вашем компьютере (как в Linux и macOS). В противном случае это относительное имя текущего каталога.

Следующие сегменты являются частью одной программы, которую мы разделили 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 имеет три столбца:

```
>>> 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 раньше вы никогда не создавали базы данных, примите мои поздравления. Можете вычеркнуть этот пункт из своего списка дел, которые в жизни обязательно нужно реализовать.

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>
```

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 тот, в котором использовался SQLite DB-API. Единственное преимущество подобного подхода состоит в том, что нам не нужно импортировать драйвер — SQLAlchemy сам определит драйвер на основе строки соединения. Простое изменение строки соединения позволит перенести код на базу данных другого типа. Еще один плюс SQLAlchemy заключается в наличии *пула соединений*, о котором вы можете прочитать в документации (https://docs.sqlalchemy.org/en/13/core/pooling.html).

Язык выражений SQL. Следующий уровень SQLAlchemy — 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, и может оказаться полезным промежуточным решением для приложений, работающих с реляционными базами данных.

Рассмотрим создание и наполнение таблицы `zoo`. И снова 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 столбца, в методе `Table()`тоже 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`. Функция `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)]
```

Object-Relational Mapper (ORM). В предыдущем подразделе объект `zoo` был 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 придерживаться таким образом принципов работы с Python, но при этом использовать реляционную базу данных.

Мы определим класс `Zoo` и свяжем его с ORM. На этот раз мы укажем SQLite использовать файл `zoo.db` так, чтобы мы могли убедиться в работе ORM.

Как и в предыдущих двух статьях, следующие фрагменты являются частью одной программы, разделенной пояснениями. Не переживайте, если чего-то не поймете. В документации к SQLAlchemy содержатся все необходимые подробности — работа с SQLALchemy может оказаться довольно сложной.

Я просто хочу, чтобы вы поняли, как много придется поработать, и чтобы сами могли решить, какой из подходов, рассмотренных в этой главе, подходит вам 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 со столбцами 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)
```

Добавить данные в таблицу можно путем создания объектов Python. ORM управляет данными изнутри:

```
>>> first = ('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 перенести нас в страну SQL. Создаем сессию для коммуникации 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 в этом убедиться:

```
$ 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
```

Целью этого статьи было показать, что такое ORM и как оно работает на высоком уровне. Автор SQLAlchemy написал полное руководство к нему (http://bit.ly/obj-rel-tutorial). После прочтения статьи определитесь, какой из следующих уровней наиболее подходит для ваших нужд:

- простой DB-API, показанный ранее в этой главе в подразделе «SQLite»;
- движок SQLAlchemy;
- язык выражений SQLAlchemy;
- SQLAlchemy ORM.

Естественным выбором выглядит ORM, что позволит избежать всех сложностей SQL. Стоит ли им пользоваться? Некоторые люди считают, что ORM следует избегать, а другие полагают, что критика незаслуженна. Кто бы ни был прав, ORM — это абстракция, а все абстракции в какой-то момент разрушаются — они допускают утечки памяти (https://www.joelonsoftware.com/2002/11/11/the-law-of-leaky-abstractions/). Если ORM не делает того, что вам нужно, вы должны понять, как он работает, а затем разобраться, как это исправить с помощью SQL. Перефразируя интернет-мем: «Столкнувшись с проблемой, некоторые люди думают: “Точно, обращусь-ка я к ORM!” Теперь у них две проблемы».

Используйте ORM для простых приложений или приложений, которые довольно точно отображают данные в таблицах базы данных. Но если приложение кажется простым, то вам, возможно, стоит использовать простой SQL (или язык выражений SQL).

Другие пакеты для работы с базами данных

Если вы ищете инструменты Python, которые позволяют работать со многими базами данных и имеют больше возможностей, чем простой DB-API, но меньше, чем SQLAlchemy, рассмотрите следующие варианты:

- `dataset` (https://dataset.readthedocs.io/en/latest/). Его девиз — «Базы данных для ленивых». Он создан на основе SQLAlchemy и предоставляет простой ORM для хранилищ SQL, JSON и CSV;
- `records` (https://pypi.org/project/records/). Его разработчики считают, что это «SQL для людей». Он поддерживает только запросы SQL, используя SQLAlchemy для обработки проблем с диалектами SQL, пулами соединений и т. д., и интегрируется с `tablib` (упоминается в подразделе «Tablib» на с. 338), что позволяет экспортировать данные в форматах CSV, JSON и многих других.

Хранилища данных NoSQL

Реляционные таблицы имеют форму прямоугольника, но поступающие данные могут иметь разную форму и их размещение требует больших усилий. Это похоже на проблему квадратного колышка и круглой дырки.

Отдельные нереляционные базы данных позволяют более гибко определять данные, работают с очень крупными наборами и поддерживают пользовательские операции с данными. Такие базы данных называют NoSQL (раньше это означало *«не SQL»*, теперь же расшифровка звучит как *«не только SQL»*).

Простейшей вариацией баз данных NoSQL являются хранилища *ключей-значений*. О некоторых из них, которые попали в рейтинг популярности (https://db-engines.com/en/ranking/key-value+store), поговорим в следующих подразделах.

Семейство dbm

Форматы `dbm` существовали задолго до того, как появился *NoSQL*. Они представляют собой простые хранилища, работающие по принципу «ключ — значение». Их часто встраивают в приложения вроде браузеров, чтобы поддерживать различные настройки. База данных dbm похожа на обычный словарь в следующих отношениях:

- вы присваиваете значение ключу, и оно автоматически сохраняется в базе данных на диске;
- вы можете получить ключ по его значению.

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, действительно ли наши данные 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». Часто его размещают перед базой данных или используют для хранения данных сессии веб-сервера. Вы можете загрузить версии для Linux, macOS (http://bit.ly/install-osx) и Windows (http://bit.ly/memcache-win). Если хотите попробовать запустить примеры, показанные в этом подразделе, вам понадобятся запущенный сервер `memcached` и драйвер Python.

Существует множество драйверов Python. Тот, что работает с Python 3, называется `python3-memcached` (https://github.com/eguven/python3-memcached). Установить его можно с помощью такой команды:

```
$ pip install python-memcached
```

Чтобы его использовать, подключитесь к серверу memcached, после чего вы сможете:

- устанавливать и получать значения ключей;
- увеличивать и уменьшать значения;
- удалять ключи.

Ключи и значения, хранимые в базе, *неустойчивы* и могут исчезать. Это происходит из-за того, что memcached является сервером кэша, а не базой данных. 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 сервером на том же компьютере:

```
>>> 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*. Он работает с ключами и их значениями, но значения имеют гораздо больше возможностей, чем в других хранилищах. 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, по сравнению со строками, структуры данных.

Типы данных Redis близки к типам данных Python, и сервер Redis может быть полезным посредником для обмена данными между приложениями. Мне это кажется настолько важным, что я посвящу данной теме небольшой фрагмент книги.

Исходный код драйвера Python `redis-py` и тесты находятся на GitHub (https://github.com/andymccurdy/redis-py), документация по нему находится по адресу http://bit.ly/redis-py-docs. Драйвер устанавливается 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`, вы сможете запустить программы, описанные в следующих подразделах.

Строки. Ключ, имеющий одно значение, является *строкой* Redis. Простые типы данных Python автоматически преобразуются. Подключимся к серверу Redis, расположенному на определенных 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 `0` означает начало списка, `-1` — конец):

```
>>> 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 15 показано, как использовать списки Redis и механизм *публикации — подписки*, чтобы реализовать очереди задач.

Хеши. Хеши Redis похожи на словари в Python, но содержат только строки, поэтому мы можем 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, в чем вы сможете убедиться 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 разницу 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 *упорядоченное множество*, или *zset*. Он представляет собой набор уникальных значений, но с каждым значением связан *счетчик* с плавающей точкой. Вы можете получить доступ к элементу по его значению или по счетчику. Упорядоченные множества применяются как:

- списки лидеров;
- списки вторичных индексов;
- временные ряды, 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)]
```

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. 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 `epoch` Unix. Истечение срока действия ключа полезно для поддержания свежести кэшей и ограничения количества сеансов входа в систему. Рассмотрим аналогию: в холодильнике, расположенном за стойками с молоком в вашем продуктовом магазине, работники избавляются от галлонов молока, когда у тех истекает срок годности.

Документоориентированные базы данных

Документоориентированные базы данных — это базы данных формата NoSQL, которые хранят данные с разными полями. В сравнении с реляционной таблицей (прямоугольной формы, с одинаковыми столбцами для каждой строки) данные, хранящиеся в этих таблицах, являются ragged — могут содержать разные поля (столбцы) в каждой строке, а также могут иметь вложенные поля. Можно обрабатывать такие данные с помощью словарей и списков или же сохранять их в файлах JSON. Для сохранения таких данных в реляционную таблицу нужно определить все возможные столбцы и использовать значения null для отсутствующих данных.

ODM может расшифровываться как Object Data Manager или Object Document Mapper. ODM является документоориентированным аналогом ORM — реляционных

баз данных. Некоторые популярные (https://db-engines.com/en/ranking/document+store) документоориентированные базы данных и инструменты (драйверы и ODM) перечислены в табл. 16.6.

Таблица 16.6. Документоориентированные базы данных

База данных	Python API
Mongo (https://www.mongodb.com/)	tools (https://api.mongodb.com/python/current/tools.html)
DynamoDB (https://aws.amazon.com/ru/dynamodb/)	boto3 (https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/GettingStarted.Python.html)
CouchDB (https://couchdb.apache.org/)	couchdb (https://couchdb-python.readthedocs.io/en/latest/index.html)

PostgreSQL может выполнять некоторые задачи, которые выполняют и документоориентированные базы данных. Его расширения позволяют избежать определенных минусов реляционных баз данных, но сохраняют при этом такие особенности, как транзакции, валидация данных и внешние ключи: 1) многомерные массивы (https://www.postgresql.org/docs/9.1/arrays.html) позволяют хранить более одного значения в ячейке таблицы; 2) jsonb (https://www.postgresql.org/docs/9.6/datatype-json.html) позволяет сохранить в ячейке данные в формате JSON с полным индексированием и запросами.

Базы данных временных рядов

Данные временных рядов могут быть собраны в фиксированные (например, метрики производительности компьютера) или случайные интервалы времени. Это привело к появлению множества методов их хранения (https://db-engines.com/en/ranking/time+series+dbms). Методы, поддерживаемые Python, перечислены в табл. 16.7.

Таблица 16.7. Временные базы данных

База данных	Python API
InfluxDB (https://www.influxdata.com/)	influx-client (https://pypi.org/project/influx-client/)
kdb+ (https://kx.com/)	PyQ (https://code.kx.com/v2/interfaces/pyq/)
Prometheus (https://prometheus.io/)	prometheus_client (https://github.com/prometheus/client_python/blob/master/README.md)
TimescaleDB (https://www.timescale.com/)	PostgreSQL-клиенты
OpenTSDB (http://opentsdb.net/)	potsdb (https://pypi.org/project/potsdb/)
PyStore (https://github.com/ranaroussi/pystore)	PyStore (https://pypi.org/project/PyStore/)

Графовые базы данных

Последний вид данных, для которых нужна собственная база данных, представляет собой графы: *узлы* (данные), соединенные *ребрами* (отношениями). Отдельный пользователь Twitter может быть узлом с ребрами, ведущими к другим пользователям, на которых *подписан он* или которые *подписаны на него*.

Графовые данные стали появляться чаще по мере роста соцсетей, где связи так же важны, как и содержимое. Некоторые популярные (https://db-engines.com/en/ranking/graph+dbms) графовые базы данных приведены в табл. 16.8.

Таблица 16.8. Графовые базы данных

База данных	**Python API**
Neo4J (https://neo4j.com/)	py2neo (https://py2neo.org/v3/)
OrientDB (https://orientdb.com/)	pyorient (https://orientdb.com/docs/last/PyOrient.html)
ArangoDB (https://www.arangodb.com/)	pyArango (https://github.com/ArangoDB-Community/pyArango)

Другие серверы NoSQL

Серверы NoSQL, перечисленные здесь, могут работать с данными, объем которых превышает объем доступной памяти, — многие из них требуют использования нескольких компьютеров. В табл. 16.9 показаны наиболее популярные серверы и их библиотеки Python.

Таблица 16.9. Базы данных NoSQL

База данных	**Python API**
Cassandra (http://cassandra.apache.org/)	pycassa (https://github.com/pycassa/pycassa)
CouchDB (https://couchdb.apache.org/)	couchdb-python (https://github.com/djc/couchdb-python)
HBase (http://hbase.apache.org/)	happybase (https://github.com/python-happybase/happybase)
Kyoto (https://fallabs.com/kyotocabinet/)	kyotocabinet (https://fallabs.com/kyotocabinet/pythondoc/)
MongoDB (https://www.mongodb.com/)	mongodb (https://api.mongodb.com/python/current/)
Pilosa (https://www.pilosa.com/)	python-pilosa (https://github.com/pilosa/python-pilosa)
Riak (https://riak.com/riak/)	riak-python-client (https://github.com/basho/riak-python-client)

Полнотекстовые базы данных

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. Популярные примеры таких баз данных с открытым исходным кодом и их Python API представлены в табл. 16.10.

Таблица 16.10. Полнотекстовые базы данных

Сайт	**Python API**
Lucene (https://lucene.apache.org/)	pylucene (https://lucene.apache.org/pylucene/)
Solr (https://lucene.apache.org/solr/)	SolPython (https://cwiki.apache.org/confluence/display/solr/SolPython)
ElasticSearch (https://www.elastic.co/)	elasticsearch (https://elasticsearch-py.readthedocs.io/en/master/)
Sphinx (http://sphinxsearch.com/)	sphinxapi (https://code.google.com/p/sphinxsearch/source/browse/trunk/api/sphinxapi.py)
Xapian (https://xapian.org/)	xappy (https://code.google.com/archive/p/xappy/)
Whoosh (https://bitbucket.org/mchaput/whoosh/wiki/Home)	Написан на Python, уже содержит API

Читайте далее

В предыдущей главе рассматривался вопрос выполнения разных фрагментов кода за разные промежутки времени *(конкурентность)*. Следующая глава посвящена перемещению данных в пространстве *(сети)* — его можно использовать для написания конкурентного кода и для многого другого.

Упражнения

16.1. Сохраните следующие несколько строк в файл `books.csv`. Обратите внимание на то, что, если поля разделены запятыми, вам нужно заключить в кавычки поле, содержащее запятую:

```
author,book
J R R Tolkien,The Hobbit
Lynne Truss,"Eats, Shoots & Leaves"
```

16.2. 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?

16.3. 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
```

16.4. Используйте модуль `sqlite3`, чтобы создать базу данных SQLite `books.db` и таблицу `books`, содержащую следующие поля: `title` (текст), `author` (текст) и `year` (целое число).

16.5. Считайте данные из файла `books.csv` и добавьте их в таблицу `book`.

16.6. Считайте и выведите на экран столбец `title` таблицы `book` в алфавитном порядке.

16.7. Считайте и выведите на экран все столбцы таблицы `book` в порядке публикации.

16.8. Используйте модуль `sqlalchemy`, чтобы подключиться к базе данных `sqlite3 books.db`, которую вы создали в упражнении 16.4. Как и в упражнении 16.6, считайте и выведите на экран столбец `title` таблицы `book` в алфавитном порядке.

16.9. 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`.

16.10. Увеличьте поле `count` хеша `test` и выведите его на экран.

ГЛАВА 17

Данные в пространстве: сети

> Время — то, что удерживает все от того, чтобы случаться сразу. Пространство — то, что удерживает все от того, чтобы случаться со мной.
>
> *Цитаты о времени*[1]

Из главы 15 вы узнали о *конкурентности*: способе выполнять больше одного дела за раз. Теперь мы попробуем решать задачи более чем в одном месте. В этом нам помогут *распределенные вычисления* и *сети*. Существует несколько хороших причин бросить вызов пространству и времени.

- *Производительность*. Ваша цель заключается в том, чтобы быстрые компоненты оставались занятыми, а не ждали, пока отработают медленные.
- *Устойчивость*. Один в поле не воин, поэтому стоит дублировать задачи для того, чтобы обойти проблемы, связанные с отказом аппаратного и программного обеспечения.
- *Простота*. Лучше всего разбивать сложные задачи на множество простых, которые легче создавать, понимать и исправлять.
- *Масштабируемость*. Увеличивайте количество серверов, чтобы справляться с нагрузкой, и уменьшайте его для экономии денег.

В этой главе мы сначала рассмотрим примитивные концепции работы с сетями, а затем поднимемся на самый высокий уровень. Начнем с модели TCP/IP и сокетов.

TCP/IP

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 существо-

[1] http://bit.ly/wiki-time.

вания слоев — позволить быть 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 IP (Internet Protocol — интернет-протокол), на котором определяется, как адресуются локации сети и перемещаются *пакеты* (фрагменты) данных. На слое, расположенном выше IP, два протокола описывают, как перемещать байты между локациями.

- *UDP (User Datagram Protocol — протокол датаграмм пользователя)*. Служит для обмена небольшим объемом данных. *Датаграмма* — это небольшое сообщение, которое отправляется целиком; оно похоже на открытку.
- *TCP (Transmission Control Protocol — протокол управления передачей)*. 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, то она выглядела 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.
О'кей, теперь я отправлю тебе шутку про TCP.
О'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*. При подключении к Интернету у вашей машины также появится *публичный* IP. Если же вы используете домашний компьютер, то он будет скрыт за оборудованием, таким как кабельный модем или роутер. Вы можете запускать интернет-протоколы даже между процессами на одной машине.

Большая часть Интернета, с которой мы будем взаимодействовать, — Всемирная паутина, серверы баз данных и т. д. — основана на протоколе TCP, функционирующем поверх протокола IP, сокращенно — TCP/IP. Сначала рассмотрим самые простые интернет-сервисы. Затем перейдем к общим шаблонам для работы с сетями.

Сокеты

Если вы хотите узнать, как все работает, то этот подраздел 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 наподобие ZeroMQ, однако 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. Клиент должен знать эти два значения, чтобы доставить сообщение и получить ответ.

В следующем коде клиента и сервера элемент `address` — это кортеж вида (*адрес*, *порт*). Данный элемент 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 — клиент, а вторая — сервер. В каждой из них мы выведем на экран время и откроем сокет. Сервер будет слушать подключения к своему сокету, а клиент — писать данные в свой сокет, который передаст сообщение на сервер.

В примере 17.1 показана первая программа, `udp_server.py`.

Пример 17.1. 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). Параметр `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.) Параметр `SOCK_DGRAM` означает, что мы будем отправлять и получать датаграммы — другими словами, станем использовать UDP.

Теперь сервер просто ждет прихода датаграммы (`recvfrom`). Когда она 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` (пример 17.2).

Пример 17.2. 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 данные, а затем получает их, в то время как сервер сначала получает.

Запустим сервер в отдельном окне. Он выведет приветственное сообщение и будет спокойно ждать 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, отправит данные (слово *Hey* в байтовом формате), покажет 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 наподобие этого 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, однако он не указал свой номер порта, поэтому тот был автоматически присвоен системой — в данном 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, то они могут прийти в неправильном порядке или вообще не появиться. Этот протокол быстр, легок, не создает соединений, однако ненадежен. Он полезен в том случае, когда нужно отправить пакеты быстро и не страшны их потери, которые происходят время от времени — например, при использовании технологии VoIP (voice over IP).

Вышесказанное приводит нас к протоколу TCP (Transmission Control Protocol — протокол управления передачей). Он используется для более продолжительных соединений, таких как соединения с Интернетом. TCP доставляет данные в том порядке, в котором те были отправлены. Если возникают какие-то проблемы, то он пытается отправить данные снова. Из-за этого протокол TCP работает немного медленнее, чем UDP, но, как правило, он предпочтительнее, если вам нужно получить все пакеты в правильном порядке.

Первые две версии веб-протокола HTTP были основаны на TCP, но HTTP/3 основан на протоколе QUIC (https://oreil.ly/Y3Jym), который использует UDP. Поэтому при выборе между UDP и TCP следует учитывать множество факторов.

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 в примере 17.3.

Пример 17.3. tcp_client.py:

```
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()` с целью установить поток. Нам не нужно было делать это для UDP, поскольку каждая датаграмма после отправки предоставлялась сама себе.

Как показано в примере 17.4, файл `tcp_server.py` также отличается от своего собрата, работающего с UDP.

Пример 17.4. tcp_server.py

```
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 байт.

Как и раньше, запустим сервер, а затем клиент и понаблюдаем. 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: чтобы ответить, сервер TCP вызвал метод `client.sendall()`, а в предыдущем примере — `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 нет гарантии, 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-либо разделитель.
- Поскольку сообщения являются байтами, а не текстовыми строками Unicode, вам придется использовать тип `bytes`. Более подробную информацию об этом типе см. в главе 12.

Если после всего этого вас все еще восхищает программирование сокетов, то вам стоит посетить ресурс Python socket programming HOWTO (http://bit.ly/socket-howto), чтобы получить более подробную информацию.

scapy

Иногда вам нужно погрузиться в поток данных, путешествующих по сети. Возможно, вы хотите отладить веб-API или отследить некую проблему безопасности. Библиотека `scapy` и программа предоставляют предметно-ориентированный язык для создания и анализа пакетов в Python. Работать с ней гораздо проще, чем писать и отлаживать программы, написанные на языке C.

Стандартная команда для установки выглядит так: `pip install scapy`. Документация (https://scapy.readthedocs.io/) этой библиотеки довольно обширна. Если вы используете такие инструменты, как `tcpdump` или `wireshark`, для исследования проблем, связанных с протоколом TCP, то вам следует обратить внимание на `scapy`. Наконец, не следует путать библиотеки `scapy` и `scrapy`; последняя рассмотрена в разделе «Поиск и выборка данных» главы 18.

Netcat

Еще один инструмент для проверки сетей и портов — Netcat (https://oreil.ly/K37H2), название которого зачастую сокращают до `nc`. Рассмотрим пример соединения HTTP с сайтом Google, где запросим базовую информацию о главной странице:

```
$ $ nc www.google.com 80
HEAD / HTTP/1.1

HTTP/1.1 200 OK
Date: Sat, 27 Jul 2019 21:04:02 GMT
...
```

В следующей главе, в подразделе «Тестируем с помощью telnet» раздела «Веб-клиенты», вы увидите пример, в котором делается то же самое.

Паттерны для работы с сетями

Сетевые приложения можно создать на основе некоторых простых паттернов.

- Самым распространенным является *«Запрос — ответ»*, также известный как *«Клиент — сервер»*. Работает синхронно: клиент ожидает ответа сервера. Вы видели множество примеров его использования в этой книге. Ваш браузер — это также клиент, делающий HTTP-запрос веб-серверу, который возвращает ответ.
- Еще один распространенный паттерн — *«Разветвление на выходе»*: вы отправляете данные любому доступному работнику из пула процессов. Примером служит веб-сервер, расположенный за балансировщиком нагрузки.
- Противоположностью этого паттерна является *«Разветвление на входе»*: вы принимаете данные из одного или нескольких источников. В качестве примера можно привести регистратор, который принимает текстовые сообщения от нескольких процессов и записывает их в единый файл журнала.
- Еще один паттерн похож на радио- или телепередачи, он называется *«Публикация — подписка»* или *pub — sub*. В рамках этого паттерна публикатор рассылает данные. В простой системе их получат все подписчики. Однако зачастую они могут указать, что заинтересованы только в определенных типах данных (подобные типы часто называются *темами*), и публикатор будет отправлять лишь эту информацию. Поэтому, в отличие от паттерна «Разветвление на входе», заданный фрагмент данных может получить более чем один подписчик. Если на тему никто не подписался, то данные будут проигнорированы.

Рассмотрим некоторые примеры использования паттерна «Запрос — ответ», а затем и паттерна «Публикация — подписка».

Паттерн «Запрос — ответ»

Этот паттерн самый распространенный. Вы запрашиваете данные о DNS, сети или электронной почте у соответствующих серверов, и они либо возвращают ответ, либо сообщают о проблеме.

Я только что показал, как выполнять простые запросы с помощью протоколов UDP и TCP, но создать приложение для работы с сетями на уровне сокетов довольно сложно. Посмотрим, сможет ли нам помочь ZeroMQ.

ZeroMQ

ZeroMQ — библиотека, а не сервер. 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 паттернов. Питер Хинтдженс создал печатную версию (*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 выбирать один из нескольких языков для каждого примера. Можно даже выбрать примеры для Python (http://bit.ly/zeromq-py). В этой главе я покажу 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 паттернов. Основные «детальки Lego», представленные в следующем списке, являются типами сокетов ZeroMQ, которые из-за превратностей судьбы выглядят как паттерны 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 паттерн — это одна пара «Запрос — ответ». Он является синхронным: один сокет создает запрос, а затем другой сокет отвечает на него. Сначала рассмотрим код ответа (сервера) `zmq_server.py`, показанный в примере 17.5.

Пример 17.5. 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 сопровождает состояние. Далее создаем сокет ZeroMQ, имеющий тип `REP` (получено от REPly — ответ). Вызываем метод `bind()`, чтобы заставить его слушать определенный IP-адрес и порт. Обратите внимание: они указаны в строке, `'tcp://localhost:6789'`, а не в кортеже, как это было в случае с простыми сокетами.

Код в данном примере продолжает получать запросы от отправителя и посылать ответы. Эти сообщения могут быть очень длинными — ZeroMQ обработает детали.

В примере 17.6 показан код клиента, `zmq_client.py`. Он имеет тип `REQ` (получено от REQuest — запрос), в нем вызывается метод `connect()`, а не `bind()`:

Пример 17.6. zmq_client.py

```
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 чередующиеся 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 давали такого указания серверу, поэтому он будет ожидать новых сообщений. При повторном запуске клиент выведет те же пять строк, сервер тоже выведет их. Если вы не завершите процесс `zmq_server.py` и попробуете запустить еще один, то 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 трансформации их в байтовые строки и обратно. Если ваши сообщения имеют другие типы данных, то можете использовать библиотеку, такую как MessagePack (http://msgpack.org/).

Даже этот простой паттерн 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; как раз из-за этого в его имени есть буква Q. Здесь она расшифровывается как Queue — «очередь», M — Message («сообщение»), а Zero («ноль») означает, что ему не нужны посредники.

Несмотря на то что ZeroMQ не предоставляет никаких центральных посредников (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, а тот, в свою очередь, связывается с подключенным к нему сокетом REP (рис. 17.1). Процесс похож на то, как несколько браузеров связываются с прокси-сервером, расположенным перед фермой веб-серверов. Это позволяет 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 ответы будут возвращаться по правильному адресу.

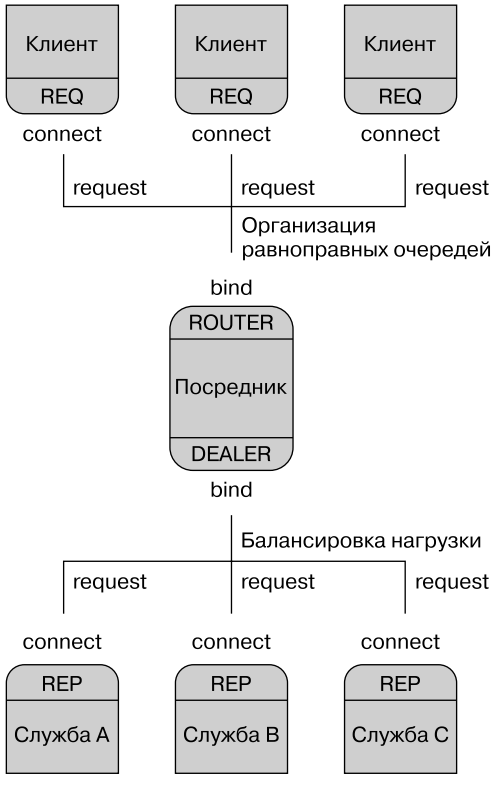

Рис. 17.1. Использование посредника для соединения с несколькими клиентами и серверами

Еще один сетевой паттерн называется *«Вентилятор»*, в его рамках используются сокеты PUSH — для перепоручения асинхронных задач, и сокеты PULL — для сбора результатов.

Последняя значимая особенность ZeroMQ — 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.

Последнее соединение, `inproc`, — способ передать данные между потоками, избежав блокировок; является альтернативой примеру работы с потоками, показанному в подразделе «Потоки» на с. 311.

После использования ZeroMQ вы, возможно, больше не захотите возвращаться к написанию чистого кода для сокетов.

Другие инструменты обмена сообщениями

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 увидите в подразделе «Apache» на с. 411, также поддерживает проект ActiveMQ (https://activemq.apache.org/), который включает несколько интерфейсов Python, использующих простой текстовый протокол STOMP (https://oreil.ly/a3h_M);
- популярна также библиотека RabbitMQ (http://www.rabbitmq.com/), вы можете прочесть онлайн-руководство для нее (http://bit.ly/rabbitmq-tut);
- NATS (http://www.nats.io/) — это быстрая система обмена сообщениями, написанная на Go.

Паттерн «Публикация — подписка»

Данный паттерн 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 данный тип. Поэтому то или иное 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 уже видели Redis в главе 16, он использовался как сервер структур данных. Однако он также содержит систему публикации — подписки. Публикатор создает сообщения, имеющие тему и значение, а подписчики указывают, какие темы хотят получать.

В примере 17.7 показан публикатор `redis_pub.py`.

Пример 17.7. 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.

В примере 17.8 показан подписчик `redis_sub.py`.

Пример 17.8. 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'`, то это значит, что он был отправлен публикатору и совпадает с нашими критериями. Ключ `'channel'` — тема (порода кота), а ключ `'data'` содержит сообщение (шляпа).

Если вы сначала запустите публикатор, который никто не станет слушать, то он будет похож на мима, упавшего в лесу (издаст ли он звук?), поэтому сначала запустим подписчик:

```
$ python redis_sub.py
```

Далее 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
```

Подписчик 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 отдали подписчику указание завершить работу, поэтому он все еще ждет сообщений. Если вы перезапустите публикатор, то он 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, то оно пропадет с сервера Redis. Но при наличии подписчиков сообщение останется на сервере, пока все они не получат его.

ZeroMQ

У ZeroMQ нет центрального сервера, поэтому каждый публикатор пишет всем подписчикам. Перепишем наш пример для ZeroMQ. Публикатор `zmq_pub.py` выглядит так, как показано в примере 17.9.

Пример 17.9. zmq_pub.py

```
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` (пример 17.10).

Пример 17.10. 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 ' '. Не сделав этого, вы не получите ничего.

Обратите внимание: в публикаторе мы вызываем метод `send_multipart()`, а в подписчике — `recv_multipart()`. Это позволяет 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 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 «Публикации — подписки»

Вам могут пригодиться следующие ссылки Python.

- RabbitMQ — широко известный посредник сообщений, Python API для него называется `pika`. Обратитесь к тематической документации (http://pika.readthedocs.org/) и руководству по «Публикации — подписке» (http://bit.ly/pub-sub-tut).

- Перейдите в окно поиска PyPi (https://pypi.python.org/) и введите `pubsub`, чтобы найти пакеты для Python, такие как `pypubsub` (http://pubsub.sourceforge.net/).
- Протокол PubSubHubbub (https://code.google.com/p/pubsubhubbub) позволяет подписчикам зарегистрировать функции обратного вызова для публикаторов.
- NATS (https://nats.io/) — это быстрая система обмена сообщениями с открытым исходным кодом, которая поддерживает паттерны «Публикация — подписка», «Запрос — ответ» и очереди.

Интернет-сервисы

Python имеет широкий набор инструментов для работы с сетями. В следующих подразделах мы рассмотрим способы автоматизации наиболее популярных интернет-сервисов. В сети доступна полная официальная документация (http://bit.ly/py-internet).

Доменная система имен

Компьютеры имеют числовые IP-адреса, например 85.2.101.94, однако имена мы запоминаем лучше, чем числа. Доменная система имен (Domain Name System, DNS) — критически важный интернет-сервис, который преобразует IP-адреса в имена и обратно с помощью распределенной базы данных. Когда вы используете браузер и внезапно видите сообщение наподобие looking up host, вы, возможно, потеряли соединение с Интернетом, и первым предположением должен стать сбой 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, 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 лишь 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 сервисами IANA и связаны с именами сервисов. Например, HTTP имеет имя `http`, ему присвоен номер порта TCP 80.

Эти функции преобразуют имена сервисов в номера портов и наоборот:

```
>>> import socket
>>> socket.getservbyname('http')
80
>>> socket.getservbyport(80)
'http'
```

Модули Python для работы с электронной почтой

Стандартная библиотека Python содержит следующие модули для работы с электронной почтой:

- `smtplib` (https://oreil.ly/_kF6V) — для отправки сообщений по электронной почте с помощью простого протокола передачи почты (Simple Mail Transfer Protocol, SMTP);
- `email` (https://oreil.ly/WVGbE) — для создания и анализа сообщений электронной почты;
- `poplib` (https://oreil.ly/xiJT7) — для чтения электронной почты с помощью протокола почтового отделения, версия 3 (Post Office Protocol 3, POP3);
- `imaplib` (https://oreil.ly/wengo) — для чтения электронной почты с помощью протокола доступа к электронной почте (Internet Message Access Protocol, IMAP).

Если вы хотите написать собственный SMTP-сервер на Python, то попробуйте `smtpd` (https://oreil.ly/JkLsD) или его новую асинхронную версию — `aiosmtpd` (https://aiosmtpd.readthedocs.io/).

Другие протоколы

Используя стандартный модуль `ftplib` (http://bit.ly/py-ftplib), вы можете перемещать байты с помощью протокола передачи файлов (File Transfer Protocol, FTP). Несмотря на свой возраст, он все еще хорошо работает.

Вы видели, как эти модули применяются повсеместно в разных местах данной книги. Взгляните также на документацию, касающуюся поддержки интернет-протоколов в стандартной библиотеке (http://bit.ly/py-internet).

Веб-сервисы и API

Поставщики информации всегда имеют сайт, однако он предназначен для человеческих глаз, а не для машин. Если данные опубликованы только на нем, то любой, кто хочет получить к ним доступ, должен писать краулер (это показано в разделе

«Поиск и выборка данных» на с. 424) и переписывать их после каждого изменения формата. Обычно данная процедура утомительна. В противоположность ей если сайт предлагает API для своих данных, то эти данные становятся доступными для клиентских программ. API меняются реже, чем макеты веб-страниц, поэтому и изменения в клиентах распространены меньше. Быстрый чистый конвейер также позволяет упростить создание гибридных приложений — комбинаций, которые не предвиделись, но могут быть полезны и даже прибыльны.

Простейший API — веб-интерфейс, который предоставляет данные в структурированном формате, таком как JSON или XML (однако не в текстовом и не в формате HTML). API может быть минимальным или полнофункциональным RESTful API (данное понятие рассматривается в разделе «REST API» на с. 424), это позволит найти еще один выход для байтов, работающих без устали.

В самом начале книги вы видели веб-API — этот интерфейс запрашивал старую копию сайта из Internet Archive.

API особенно полезны для получения данных с популярных сайтов социальных медиа, таких как 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 служить для ограничения трафика запросов к серверам.

У следующих брендов имеются интересные сервисы API:

- New York Times (http://developer.nytimes.com/);
- Twitter (https://python-twitter.readthedocs.io/);
- 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 главе 21, примеры других API — в главе 22.

Сериализация данных

Как вы видели в главе 16, такие форматы данных, как XML, JSON и YAML, представляют собой способы хранения структурированных текстовых данных. Сетевым приложениям необходимо обмениваться данными с другими программами. Преобразование между данными в памяти и последовательностями байтов (в частности, перед их отправкой другому клиенту) называется *сериализацией* или *маршаллингом*. JSON — это популярный формат сериализации, особенно часто используемый в RESTful-системах, но с его помощью нельзя выразить все типы данных, используемые в Python. Кроме того, как текстовый формат он выглядит более многословным в отличие от ряда бинарных методов сериализации. Рассмотрим несколько подходов, с которыми вы можете столкнуться.

Сериализация с помощью pickle

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)
```

Модуль `pickle` работает и с вашими классами и объектами. Мы определим небольшой класс, который называется `Tiny` и возвращает слово `'tiny'`, когда 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'
```

Строка `pickled` — это обработанная `pickle` бинарная строка, созданная из объекта `obj1`. Мы преобразовали ее в объект `obj2` с целью сделать копию объекта `obj1`. Используйте функцию `dump()`, чтобы `pickle` сохранил данные в файл, и функцию `load()`, чтобы `pickle` загрузил данные из файла.

Модуль `multiprocessing` использует `pickle` для обмена данными между процессами.

Если модуль `pickle` не может сериализовать ваш формат данных, то с ним, возможно, справится более новый сторонний пакет `dill` (https://pypi.org/project/dill).

Поскольку модуль pickle может создавать объекты Python, к нему применимы предупреждения о безопасности, рассмотренные ранее. Не загружайте в pickle данные, которым не доверяете.

Другие форматы сериализации

Есть и другие бинарные форматы обмена данными, они обычно компактнее и быстрее, чем XML или JSON:

- MsgPack (http://msgpack.org/);
- Protocol Buffers (https://code.google.com/p/protobuf);
- Avro (http://avro.apache.org/docs/current);
- Thrift (http://thrift.apache.org/);
- Lima (https://lima.readthedocs.io/);
- Serialize (https://pypi.org/project/Serialize) — это фронтенд Python для других форматов, включающих JSON, YAML, pickle и MsgPack;
- Бенчмарк (https://oreil.ly/S3ESH) различных пакетов сериализации для Python.

Поскольку они бинарные, ни один из них не может быть изменен человеком, вооружившимся текстовым редактором.

Некоторые сторонние пакеты выполняют двухстороннюю конвертацию объектов и простых типов данных Python (что позволяет преобразовывать их в такие форматы, как JSON), а также могут выполнять *валидацию* таких категорий:

- типы данных;
- диапазоны значений;
- обязательные и необязательные данные.

Ниже представлены некоторые из этих пакетов:

- Marshmallow (https://marshmallow.readthedocs.io/en/3.0);
- Pydantic (https://pydantic-docs.helpmanual.io/) — использует подсказки для типов, поэтому требует версии Python не ниже 3.6;
- TypeSystem (https://www.encode.io/typesystem).

Эти пакеты зачастую используются вместе с веб-серверами для гарантии того, что байты, которые попали к ним по протоколу HTTP, отправятся в правильные структуры данных для дальнейшей обработки.

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 собственной машине. При этом в RPC-клиенте произойдет следующее.

1. Он преобразует аргументы вашей функции в байты.
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.
3. Затем клиент находит и вызывает локальную функцию с помощью раскодированных данных.
4. Далее он кодирует результат работы функции.
5. Наконец, отправляет закодированные байты вызывающей стороне.

Затем 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-либо способа транспортировки байтов и метода кодирования/декодирования, определяете функции сервиса и подаете сигнал *«RPC готов к работе»*. Клиенты соединяются с сервером и вызывают одну из его функций с помощью RPC.

XML 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` (пример 17.11).

Пример 17.11. 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 его же, 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 пример 17.12, в нем показывается файл `xmlrpc_client.py`.

Пример 17.12. 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 -
```

Популярные методы передачи данных — HTTP и ZeroMQ.

JSON RPC

JSON-RPC (версии 1.0 (https://oreil.ly/OklKa) и 2.0 (https://oreil.ly/4CS0r)) аналогичны XML-RPC, но в них используется JSON. Существует множество библиотек для работы с JSON-RPC, самая простая из них состоит из двух частей: клиента (https://oreil.ly/8npxf) и сервера (https://oreil.ly/P_uDr).

Они устанавливаются с помощью команд `pip install jsonrpcserver` и `pip install jsonrpclient`.

Эти библиотеки предоставляют множество альтернативных способов написать клиент (https://oreil.ly/fd412) и сервер (https://oreil.ly/SINeg). В примерах 17.13 и 17.14 я применяю встроенный в данную библиотеку сервер, который использует порт 5000 и является простейшим.

Сначала рассмотрим сервер.

Пример 17.13. jsonrpc_server.py

```
from jsonrpcserver import method, serve

@method
def double(num):
    return num * 2
```

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

Затем рассмотрим клиент.

Пример 17.14. jsonrpc_client.py

```
from jsonrpcclient import request

num = 7
response = request("http://localhost:5000", "double", num=num)
print("Double", num, "is", response.data.result)
```

Как и в случае с большинством примеров архитектуры «Клиент — сервер», рассмотренных в данной главе, сначала запустите сервер (сделайте это в отдельном окне терминала или добавьте после его имени символ &, чтобы запустить его в фоновом режиме), а затем запустите клиент:

```
$ python jsonrpc_server.py &
[1] 10621
$ python jsonrpc_client.py
127.0.0.1 - - [23/Jun/2019 15:39:24] "POST / HTTP/1.1" 200 -
Double 7 is 14
```

Если вы запустили сервер в фоновом режиме, то убейте процесс после того, как закончите работу.

MessagePack RPC

Библиотека кодирования MessagePack имеет собственную реализацию RPC (http://bit.ly/msgpack-rpc). Она устанавливается следующим образом:

```
$ pip install msgpack-rpc-python
```

Эта команда также установит `tornado`, веб-сервер, основанный на событиях, который данная библиотека использует в качестве транспорта. Как обычно, сначала рассмотрим сервер (`msgpack_server.py`, пример 17.15).

Пример 17.15. 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()
```

Класс `Services` предоставляет свои методы как сервисы RPC. Теперь запустим клиент, `msgpack_client.py` (пример 17.16).

Пример 17.16. 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))
```

Чтобы увидеть код в действии, выполним привычный порядок действий: запустим сервер и клиент в отдельных окнах терминала[1], а затем пронаблюдаем результат:

```
$ python msgpack_server.py

$ python msgpack_client.py
Double 8 is 16
```

Zerorpc

Пакет zerorpc (http://www.zerorpc.io/) написан разработчиками Docker (когда они еще назывались dotCloud). Он использует ZeroMQ и MsgPack для соединения клиентов и серверов и магическим образом предоставляет функции как конечные точки RPC.

Введите команду `pip install zerorpc`, чтобы установить его. В примерах 17.17 и 17.18 показаны клиент и сервер, работающие по принципу «запрос — ответ».

Пример 17.17. zerorpc_server.py

```
import zerorpc

class RPC():
    def double(self, num):
        return 2 * num

server = zerorpc.Server(RPC())
server.bind("tcp://0.0.0.0:4242")
server.run()
```

Пример 17.18. zerorpc_client.py

```
import zerorpc

client = zerorpc.Client()
client.connect("tcp://127.0.0.1:4242")
num = 7
result = client.double(num)
print("Double", num, "is", result)
```

Обратите внимание: клиент вызывает метод `client.double()` даже несмотря на то, что его определение отсутствует:

---

[1] Или поместим его в фоновый режим с помощью символа &.

```
$ python zerorpc_server &
[1] 55172
$ python zerorpc_client.py
Double 7 is 14
```

На сайте https://github.com/0rpc/zerorpc-python вы сможете найти еще больше примеров.

gRPC

Компания Google создала gRPC (https://grpc.io/) как портативный и быстрый способ определять и соединять сервисы. Этот пакет кодирует данные как буферы протоколов (https://oreil.ly/UINlc).

Установите те части, которые касаются Python:

```
$ pip install grpcio
$ pip install grpcio-tools
```

Документация к клиенту Python (https://grpc.io/docs/quickstart/python) очень подробная, так что здесь я приведу лишь краткий обзор его возможностей. Кроме того, вам может понравиться это отдельное руководство (https://oreil.ly/awnxO).

Чтобы использовать gRPC, нужно создать файл с расширением `.proto` для определения *сервиса* и его методов `rpc`.

Метод `rpc` похож на определение функции (в нем описываются его аргументы и возвращаемые типы) и может указывать на один из следующих паттернов работы с сетью:

- запрос — ответ (синхронно или асинхронно);
- запрос — потоковый ответ;
- потоковый запрос — ответ (синхронно или асинхронно);
- потоковый запрос — потоковый ответ.

Отдельные запросы могут быть блокирующими или асинхронными. По потоковым запросам можно итерировать.

Далее вам нужно запустить программу `grpc_tools.protoc`, чтобы создать код для клиента и сервера. gRPC позволяет выполнить сериализацию и сетевое соединение; вы добавляете свой код, характерный только для вашего приложения, в заглушки клиента и сервера.

gRPC — высокоуровневая альтернатива REST API. Этот пакет больше подходит для межсервисного взаимодействия, а REST — для публичных API.

Twirp

Twirp (https://oreil.ly/buf4x) похож на gRPC, однако его создатели заявляют, что он проще. Вы определяете файл с расширением `.proto`, как и в случае с gRPC, и twirp может сгенерировать код Python, который обрабатывает клиентскую и серверную части.

Инструменты удаленного управления

- `Salt` (http://www.saltstack.com/) написан на Python. Он создавался как способ реализовать удаленное выполнение программ, но позже вырос в полноценную платформу управления системами. Основанный на ZeroMQ вместо SSH, он может работать с тысячами серверов.
- 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, вполне сопоставим с ними. Вы можете скачать и применять его бесплатно, но поддержка и отдельные 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 обрабатывают исходную конфигурацию, развертывание и удаленное выполнение.

Работаем с большими объемами данных

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 *поиск* требовал множества механических движений дисковой головки. (Подумайте о виниловой пластинке и времени, необходимом для перемещения иголки 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 объединенных в сеть машинах, чем на отдельных. Они могли использовать алгоритмы, которые звучали просто, однако 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.

Hadoop

После того как компания Google опубликовала (https://oreil.ly/cla0d) полученные результаты, компания Yahoo! вслед за ней создала пакет с открытым исходным кодом, написанный на Java, который называется *Hadoop* (в честь игрушечного плюшевого слона, принадлежавшего сыну главного разработчика).

Здесь вступают в действие слова *«большие данные»*. Зачастую они просто означают следующее: «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. Этот пакет 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 быстрый метод — *отправка потоком с помощью Hadoop*, который работает как каналы Unix, посылая данные между программами и не требуя записи на диск после выполнения каждого 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 — написанного на Python Luigi (https://github.com/spotify/luigi).

Spark

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).

Disco

Еще одна альтернатива Hadoop — 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` — см. документацию (http://bit.ly/get-disco).

Dask

Dask (https://dask.org/) похож на Spark, однако написан на Python и широко используется в научных пакетах, таких как NumPy, Pandas и scikit-learn. Он может распространять задачи в кластерах, содержащих тысячи машин.

Для получения Dask и всех вспомогательных модулей выполните следующую команду:

```
$ pip install dask[complete]
```

Обратитесь к главе 22, чтобы увидеть связанные с нашей темой примеры *параллельного программирования*, в которых объемные структурированные вычисления распространены между несколькими машинами.

Работаем в облаках

Я действительно совсем не знаю облаков.

Джони Митчелл

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 некую плату, но вам все равно приходилось брать в аренду физические устройства и постоянно платить за пиковую нагрузку.

С увеличением количества компьютеров ошибки все более распространены. Вам нужно масштабировать сервисы горизонтально и хранить избыточные данные. Вы не можете предполагать, что сеть будет работать как одна машина. Согласно Питеру Дойчу, 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. Серверы можно сравнить с домашними животными: вы даете им имена, знаете их характер и лечите 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-либо проблема.

Вместо того чтобы создавать систему, вы можете арендовать сервер в *облаке*. В рамках этой модели обслуживание серверов — забота кого-то другого, а вы можете сконцентрироваться на своем сервисе, 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, если какой-то

показатель превысил лимит. Облака в данный момент довольно популярная тема, и корпоративные расходы на облачные компоненты резко возросли.

Самые крупные поставщики облачных сервисов:

- Amazon (AWS);
- Google;
- Microsoft Azure.

Amazon Web Services

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 объявил работникам Amazon, что с этого момента все данные и функционал должны быть доступны только через интерфейсы сетевых сервисов — не через файлы, базы данных или локальные вызовы функций. Программистам 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 сервисы. Они позаимствовали или придумали сами множество решений, включая Amazon Web Services (AWS) (http://aws.amazon.com/), которое сейчас доминирует на рынке. Официальной библиотекой Python для работы с AWS является `boto3`:

- документация (https://oreil.ly/y2Baz);
- страницы SDK (https://aws.amazon.com/sdk-for-python).

Установите ее следующим образом:

```
$ pip install boto3
```

Вы можете использовать `boto3` как альтернативу веб-страницам для управления AWS.

Google

Google часто использует Python для внутренних нужд и нанимает именитых разработчиков Python (у них какое-то время работал сам Гвидо ван Россум). На главной странице Google (https://cloud.google.com/) и странице, посвященной Python (https://cloud.google.com/python), вы можете найти подробную информацию о его сервисах.

Microsoft Azure

Microsoft наряду с Amazon и Google предлагает облачный сервис под названием Azure (https://azure.microsoft.com/). В статье Python on Azure (https://oreil.ly/Yo6Nz) можно узнать, как разрабатывать и размещать приложения Python в этом сервисе.

OpenStack

OpenStack (https://www.openstack.org/) — это бесплатное решение с открытым исходным кодом, содержащее сервисы Python и Rest API. Многие из них аналогичны сервисам, предлагаемым коммерческими облаками.

Docker

Простой стандартизированный контейнер отправки ПО произвел революцию в международной торговле. Всего несколько лет назад Docker применил название *«контейнер»* и аналогию к методу *виртуализации* с помощью малоизвестных особенностей Linux. Контейнеры гораздо легче, чем виртуальные машины, и немного тяжелее, чем virtualenvs в Python. Они позволяют упаковать приложение отдельно от других программ на одной и той же машине, общим для них будет только ядро ОС.

Чтобы установить клиентскую библиотеку Docker (https://pypi.org/project/docker), выполните следующую команду:

```
$ pip install docker
```

Kubernetes

Технология контейнеризации распространилась по всему вычислительному миру. В конечном счете возникла потребность в способе управления большим количеством контейнеров и автоматизации операций, выполняемых вручную в крупных распределенных системах:

- обход отказов;
- балансировка нагрузки;
- масштабирование.

Похоже, что лидером на рынке *управления контейнерами* становится Kubernetes (https://kubernetes.io/).

Чтобы установить его клиентскую библиотеку (https://github.com/kubernetes-client/python), выполните следующую команду:

```
$ pip install kubernetes
```

Читайте далее

Как говорят на телевидении, наш следующий гость не нуждается в представлении. Узнаем, почему Python является одним из лучших языков для работы в Интернете.

Упражнения

17.1. Используйте объект класса `socket`, чтобы реализовать сервис, сообщающий текущее время. Когда клиент отправляет на сервер строку `'time'`, верните текущие дату и время как строку ISO.

17.2. Задействуйте сокеты ZeroMQ REQ и REP, чтобы сделать то же самое.

17.3. Попробуйте сделать то же самое с помощью XMLRPC.

17.4. Возможно, вы видели эпизод телесериала *I Love Lucy*, в котором Люси и Этель работают на шоколадной фабрике. Парочка стала отставать, когда линия конвейера, направлявшая к ним на обработку конфеты, еще более ускорилась. Напишите симуляцию, которая отправляет разные типы конфет в список Redis, и клиент Lucy, делающий 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.

17.5. Используйте ZeroMQ, чтобы публиковать стихотворение из упражнения 12.4 (пример 12.1) по одному слову за раз. Напишите потребитель ZeroMQ, который будет выводить на экран каждое слово, начинающееся с гласной. Напишите другой потребитель, который станет выводить все слова, состоящие из пяти букв. Знаки препинания игнорируйте.

ГЛАВА 18

Распутываем Всемирную паутину

О, какой сложный узор мы ткем...

Вальтер Скотт. Мармион

На французско-швейцарской границе располагается CERN — Институт исследования физики частиц, где частенько сталкивают атомы друг с другом.

При этом процессе генерируется множество данных. В 1989 году английский ученый Тим Бернерс-Ли впервые внес предложение помочь распространять информацию внутри CERN, а также среди всего 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) — протокол 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.

Все это было создано на базе сети *Интернет*, который в то время был некоммерческим, им пользовались лишь несколько университетов и исследовательских организаций.

Бернерс-Ли написал первый браузер и сервер на компьютере NeXT[1]. Известность Всемирной паутины значительно возросла в 1993-м, когда группа студентов Иллинойсского университета выпустила браузер Mosaic (для Windows, Macintosh и Unix) и сервер NCSA *httpd*. Загрузив тем летом Mosaic и начав создавать сайты, я даже не догадывался, что Всемирная паутина и Интернет станут частью повсед-

[1] Компания, основанная Стивом Джобсом после его ухода из Apple.

невной жизни. В то время Интернет[1] 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 раннего 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 особенно хорош для работы в Интернете на любом из таких уровней, как:

- 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 одноименном разделе главы 17 на с. 368 данный 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 — 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 нажав ссылку на веб-странице. Очень часто для отображения сайта используются возвращаемые данные: HTML-документы, файлы JavaScript,

---

[1] Развеем старый миф. Сенатор (а впоследствии вице-президент) Эл Гор выступал в поддержку двухпартийного законодательства и сотрудничества, включая финансирование группы, создавшей Mosaic, что позволило значительно развить Интернет на раннем этапе. Он никогда не утверждал, что «изобрел Интернет»; эту фразу ему приписали его политические конкуренты, когда он баллотировался в президенты в 2000-м.

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 описанных усложнений включают в себя *cookie*, в которых сервер отправляет клиенту довольно специфическую информацию, позволяющую их распознать, когда клиент отправляет эти файлы назад.

Тестируем с помощью telnet

HTTP — протокол, основанный на тексте, поэтому вы можете вручную вводить его код во время тестирования. Древняя программа `telnet` позволяет подключиться к любому серверу и порту и вводить команды для любого сервиса, запущенного на нем. Создавать безопасные (зашифрованные) соединения с другими машинами следует с помощью `ssh`.

Запросим у любимого многими тестового сайта Google базовую информацию о его главной странице. Введем следующее:

```
$ telnet www.google.com 80
```

Если на порте 80 (на нем работает незашифрованный `http`; зашифрованный протокол `https` использует порт 443) по адресу `google.com` существует веб-сервер (я думаю, это беспроигрышный вариант), то `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, такого как HTML-файл, и возвращает его клиенту. Для первой проверки мы используем команду HTTP `HEAD`, просто получающую некую базовую информацию *о* ресурсе:

```
HEAD / HTTP/1.1
```

Добавьте дополнительный символ возврата каретки, чтобы удаленный сервер понял: вы закончили вводить команду и хотите получить ответ. Конструкция

HEAD / отправляет запрос HTTP HEAD *глагол* (команда) с целью получить информацию о главной странице (/). Вы получите ответ наподобие следующего (я обрезал некоторые длинные строки с помощью многоточий, чтобы они не вываливались за пределы страницы):

```
HTTP/1.1 200 OK
Date: Mon, 10 Jun 2019 16:12:13 GMT
Expires: -1
Cache-Control: private, max-age=0
Content-Type: text/html; charset=ISO-8859-1
P3P: CP="This is not a P3P policy! See g.co/p3phelp for more info."
Server: gws
X-XSS-Protection: 0
X-Frame-Options: SAMEORIGIN
Set-Cookie: 1P_JAR=...; expires=... GMT; path=/; domain=.google.com
Set-Cookie: NID=...; expires=... GMT; path=/; domain=.google.com; HttpOnly
Transfer-Encoding: chunked
Accept-Ranges: none
Vary: Accept-Encoding
```

Так выглядят заголовки ответов HTTP и их значения. Некоторые из них, такие как Date или Content-Type, обязательны. Другие, например Set-Cookie, используются для отслеживания вашей активности в течение нескольких посещений (мы поговорим об *управлении состоянием* чуть позже). Делая запрос HTTP HEAD, вы получаете в ответ только заголовки. Если вы задействовали команды HTTP GET или HTTP POST, то также получите данные от главной страницы (смесь HTML, CSS, JavaScript и всего прочего, что Google решит разместить на своей главной странице).

Я не хочу, чтобы вы зависли в telnet. Закрыть его можно, введя следующее:

```
q
```

Тестируем с помощью curl

Использовать telnet довольно просто, но это делается исключительно вручную. Программа curl (https://curl.haxx.se/), возможно, самый популярный веб-клиент, работающий в командной строке. Его документация включает в себя книгу *Everything Curl* (https://curl.haxx.se/book.html), доступную в форматах HTML, PDF и ebook. В этой таблице (https://oreil.ly/dLR8b) сравниваются curl и похожие инструменты. На странице скачивания (https://curl.haxx.se/download.html) перечислены все основные, а также многие узкоспециализированные платформы.

Простейший вариант использования curl — неявный запрос GET (здесь его вывод будет обрезан):

```
$ curl http://www.example.com
<!doctype html>
<html>
<head>
    <title>Example Domain</title>
    ...
```

Здесь используется HEAD:

```
$ curl --head http://www.example.com
HTTP/1.1 200 OK
Content-Encoding: gzip
Accept-Ranges: bytes
Cache-Control: max-age=604800
Content-Type: text/html; charset=UTF-8
Date: Sun, 05 May 2019 16:14:30 GMT
Etag: "1541025663"
Expires: Sun, 12 May 2019 16:14:30 GMT
Last-Modified: Fri, 09 Aug 2013 23:54:35 GMT
Server: ECS (agb/52B1)
X-Cache: HIT
Content-Length: 606
```

Если вы передаете аргументы, то можете включить их в командную строку или в файл данных. В этих примерах я использую следующее:

- URL любого сайта;
- файл `data.txt` — текстовый файл данных со следующим содержимым: `a=1&b=2`;
- файл `data.json` — файл данных JSON с таким содержимым: `{"a": 1, "b": 2}`;
- `a=1&b=2` в качестве двух аргументов.

Использование аргументов по умолчанию *(закодированных в форме)*:

```
$ curl -X POST -d "a=1&b=2" url
$ curl -X POST -d "@data.txt" url
```

Для аргументов, закодированных в JSON:

```
$ curl -X POST -d "{'a':1,'b':2}" -H "Content-Type: application/json" url
$ curl -X POST -d "@data.json" url
```

Тестируем с использованием httpie

Альтернативой `curl` в духе Python является `httpie` (https://httpie.org/).

```
$ pip install httpie
```

Чтобы выполнить запрос `POST`, сделаем шаги, аналогичные шагам для `curl`, показанным выше (`-f` — это синоним для `--form`):

```
$ http -f POST url a=1 b=2
$ http POST -f url < data.txt
```

Кодировкой по умолчанию является JSON:

```
$ http POST url a=1 b=2
$ http POST url < data.json
```

Программа `httpie` также обрабатывает заголовки HTTP, cookies, загрузки файлов, аутентификацию, перенаправление, переадресацию, SSL и т. д. Как обычно, вы можете обратиться к документации (https://httpie.org/doc).

Тестируем с помощью httpbin

Вы можете тестировать свои веб-запросы для сайта `httpbin` (https://httpbin.org/) или же загрузить и запустить сайт в локальном образе Docker:

```
$ docker run -p 80:80 kennethreitz/httpbin
```

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 11, пакет — всего лишь каталог для хранения файлов модулей):

- `http` управляет всеми деталями клиент-серверного взаимодействия HTTP:
  - `client` выполняет всю работу на стороне клиента;
  - `server` помогает написать веб-сервер Python;
  - `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.

Если вы хотите написать код, который совместим с Python 2 и Python 3, то вам следует помнить о том, что библиотека urllib значительно (https://oreil.ly/ww5_R) изменилась при переходе на Python 3. Чуть ниже, в подразделе «За пределами стандартной библиотеки: requests» на с. 407, представлена более удобная альтернатива.

Воспользуемся стандартной библиотекой с целью получить что-либо с сайта. URL в следующем примере возвращает информацию с тестового сайта:

```
>>> import urllib.request as ur
>>>
>>> url = 'http://www.example.com/'
>>> conn = ur.urlopen(url)
```

Этот небольшой фрагмент кода открыл соединение TCP/IP с удаленным веб-сервером `www.example.com`, создал запрос HTTP и получил HTTP-ответ. Ответ содержит не только данные о странице (цитату). Из официальной документации (http://bit.ly/httpresponse-docs) мы можем узнать, что `conn` является объектом класса `HTTPResponse`, содержащим несколько методов и атрибутов. Одна из наиболее важных частей ответа — *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 некую 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)* — некие проблемы на стороне клиента, например знаменитая ошибка 404 (ресурс не найден). Код 418 *(I'm a teapot)* был первоапрельской шуткой;
- *5xx (ошибка сервера)* — код 500 — общая ошибка. Вы можете встретить ошибку 502 (ошибочный шлюз), если произошел разрыв связи между веб-сервером и машинным интерфейсом.

Чтобы получить содержимое веб-страницы, используйте метод `read()` переменной `conn`. Вы получите значение типа `bytes`. Получим данные и выведем на экран первые 50 байт:

```
>>> data = conn.read()
>>> print(data[:50])

b'<!doctype html>\n<html>\n<head>\n <title>Example D'
```

Мы можем преобразовать эти байты в строку и вывести 50 первых символов:

```
>>> str_data = data.decode('utf8')
>>> print(str_data[:50])
<!doctype html>
<html>
<head>
    <title>Example D
>>>
```

Остальная часть решения включает в себя дополнительный код HTML и CSS.

Из любопытства взглянем, какие еще заголовки HTTP были отправлены нам:

```
>>> for key, value in conn.getheaders():
...     print(key, value)
...

Cache-Control max-age=604800
Content-Type text/html; charset=UTF-8
Date Sun, 05 May 2019 03:09:26 GMT
Etag "1541025663+ident"
Expires Sun, 12 May 2019 03:09:26 GMT
Last-Modified Fri, 09 Aug 2013 23:54:35 GMT
Server ECS (agb/5296)
```

```
Vary Accept-Encoding
X-Cache HIT
Content-Length 1270
Connection close
```

Помните 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, таких как `Content-Type`, и множество опциональных.

За пределами стандартной библиотеки: requests

В начале главы 1 вы увидели программу, которая получает доступ к Wayback Machine API с помощью стандартных библиотек `urllib.request` и `json`. За ним следовал другой пример, использовавший 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 https://oreil.ly/zF8cy (она довольно хорошо написана), чтобы получить более подробную информацию. Я покажу основные принципы работы с этой библиотекой в данном подразделе и буду задействовать ее на протяжении всей книги для решения задач, связанных с веб-клиентами.

Для начала нужно установить библиотеку `requests`:

```
$ pip install requests
```

Переделаем предыдущий вызов сервиса с `example.com` с помощью библиотеки `requests`:

```
>>> import requests
>>> resp = requests.get('http://example.com')
>>> resp
<Response [200]>
>>> resp.status_code
200
>>> resp.text[:50]
'<!doctype html>\n<html>\n<head>\n    <title>Example D'
```

Для демонстрации запроса JSON создадим сокращенную версию программы, которую вы увидите в конце данной главы. Вы предоставляете строку, и программа воспользуется поисковым API Internet Archive, чтобы пройтись по названиям миллиардов мультимедиа, сохраненным в этом архиве. Обратите внимание: в вызов `requests.get()`, показанный в примере 18.1, вам нужно передать только словарь `params` и библиотека `requests` обработает управляющие символы и создаст запрос.

Пример 18.1. ia.py

```
import json
import sys
```

```
import requests

def search(title):
     url = "http://archive.org/advancedsearch.php"
     params = {"q": f"title:({title})",
               "output": "json",
               "fields": "identifier,title",
               "rows": 50,
               "page": 1,}
     resp = requests.get(url, params=params)
     return resp.json()

if __name__ == "__main__":
     title = sys.argv[1]
     data = search(title)
     docs = data["response"]["docs"]
     print(f"Found {len(docs)} items, showing first 10")
     print("identifier\ttitle")
     for row in docs[:10]:
         print(row["identifier"], row["title"], sep="\t")
```

Сколько у них элементов, связанных с вендиго?

```
$ python ia.py wendigo
Found 24 items, showing first 10
identifier  title
cd_wendigo_penny-sparrow  Wendigo
Wendigo1  Wendigo 1
wendigo_ag_librivox The Wendigo
thewendigo10897gut  The Wendigo
isbn_9780843944792  Wendigo mountain ; Death camp
jamendo-060508  Wendigo - Audio Leash
fav-lady_wendigo  lady_wendigo Favorites
011bFearTheWendigo  011b Fear The Wendigo
CharmedChats112 Episode 112 - The Wendigo
jamendo-076964  Wendigo - Tomame o Dejame>
```

Первую колонку *(идентификатор)* можно использовать для того, чтобы просмотреть элемент на сайте archive.org. Вы узнаете, как это делается, в конце главы.

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 данном языке, что теперь уже становится трудно исследовать их все и сделать выбор, не говоря уже о принятии решения о том, какие из них обсудить в книге.

Веб-фреймворк предоставляет функции, позволяющие построить сайты, поэтому может решать большее количество задач, чем простой веб-сервер (HTTP). Вы встретитесь с функциями маршрутизации (URL к функции сервера), шаблонами (HTML 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, как запускать динамические части сайта на традиционном веб-сервере с помощью Python и других составляющих.

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-сервер. Если не возникло никаких проблем, то 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, например такие:

```
127.0.0.1 - - [20/Feb/2013 22:02:37] "GET / HTTP/1.1" 200 -
```

Элементы `localhost` и `127.0.0.1` являются для TCP синонимами *вашего локального компьютера*, поэтому сработают независимо от того, подключены ли вы к Интернету. Вы можете интерпретировать эти строки следующим образом:

- `127.0.0.1` — IP-адрес клиента;
- первый символ `-` — имя удаленного пользователя, если он присутствует;
- второй символ `-` — имя авторизующегося пользователя, если требуется;
- `[20/Feb/2013 22:02:37]` — дата и время доступа;
- `"GET/HTTP/1.1"` — команда, отправленная веб-серверу:
  - метод HTTP (`GET`);
  - запрошенный ресурс (`/`, верхний уровень);
  - версия HTTP (`HTTP/1.1`);
- последнее число (`200`) — код статуса HTTP, возвращенный веб-сервером.

Щелкните на любом файле. Если ваш браузер может распознать его формат (HTML, PNG, GIF, JPEG и т. д.), то 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. 20.2, а в журнале должна появиться похожая запись:

```
127.0.0.1 - - [20/Feb/2013 22:03:48] "GET /oreilly.png HTTP/1.1" 200 -
```

При наличии в этом каталоге других файлов их названия должны появиться в списке. Можете щелкнуть на одном из файлов, чтобы загрузить его. Если ваш браузер сконфигурирован так, чтобы отображать формат данного файла, то 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. Вы можете выключить его, убив его процесс нажатием Ctrl+C.

Вы не должны использовать этот простой сервер для загруженного производственного сайта. Традиционные веб-серверы, такие как Apache и NGINX, гораздо быстрее работают со статическими файлами. Кроме того, этот простой сервер не может взаимодействовать с динамическим содержимым, в отличие от более продвинутых серверов, принимающих дополнительные параметры.

Web Server Gateway Interface (WSGI)

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.

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 на 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 функционирует данный интерфейс (для этого многого и не потребуется), но знание основных принципов его функционирования может действительно помочь разработке. Такое соединение называется *синхронным* — за одним шагом следует другой.

ASGI

Я уже несколько раз упоминал, что в Python начали появляться возможности асинхронного программирования, такие как `async`, `await` и `asyncio`. ASGI (Asynchronous Server Gateway Interface) — аналог WSGI, который использует все эти возможности. В приложении B вы познакомитесь с этой темой более подробно, а также увидите примеры новых веб-фреймворков, применяющих ASGI.

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 его 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` (пример 18.2).

Пример 18.2. home.wsgi

```
import bottle

application = bottle.default_app()

@bottle.route('/')
def home():
    return "apache and wsgi, sitting in a tree"
```

На сей раз не вызывайте функцию `run()`, поскольку это запустит встроенный веб-сервер Python. Нам нужно присвоить некое 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, то нужно лишь соединить их с нашим сценарием Python. Надо добавить в файл одну строку, которая определяет сайт по умолчанию для этого сервера `apache`, но поиск данного 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, 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, если он работал, перезапустите, с целью указать ему, что следует использовать новую конфигурацию. Перейдя в браузере по адресу http://localhost/, вы 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 этого 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>
```

NGINX

Веб-сервер NGINX (http://nginx.org/) не имеет встроенного модуля Python. Вместо этого он обменивается данными с помощью отдельного сервера WSGI, такого как uWSGI или gUnicorn. Вместе они представляют собой очень быструю и удобную в конфигурации платформу для веб-разработки на Python.

Вы можете установить `nginx` с его официального сайта http://wiki.nginx.org/Install. Небольшая страница документации (https://oreil.ly/7FTPa) предоставляет инструкции, позволяющие объединить Flask, NGINX и uWSGI.

Другие веб-серверы Python

Ниже перечислены некоторые независимые WSGI-серверы, написанные на Python, которые работают как `apache` или `nginx` и используют несколько процессов и/или потоков (см. раздел «Конкурентность» главы 15 на с. 308) для обработки одновременных запросов:

- `uwsgi` (http://projects.unbit.it/uwsgi);
- `cherrypy` (http://www.cherrypy.org/);
- `pylons` (http://www.pylonsproject.org/).

А это 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 подробнее рассказывал в разделе «Конкурентность» на с. 308.

Фреймворки для работы веб-серверами

Веб-серверы обрабатывают детали работы HTTP и WSGI, но вам нужно использовать веб-*фреймворки*, чтобы написать код на Python, который будет поддерживать сайт. Так что мы немного поговорим о фреймворках, а затем вернемся к альтернативным способам обслуживания сайтов, использующих их.

Написать сайт на Python можно с помощью множества веб-фреймворков (некоторые даже могут сказать, что их слишком много). Веб-фреймворк обрабатывает как минимум запросы клиента и ответы сервера. Большинство крупных веб-фреймворков могут решать такие задачи, как:

- обработка протокола HTTP;
- аутентификация (*authn*, или «кто ты?»);
- авторизация (*authz*, или «что ты можешь сделать?»);
- создание сессии;
- получение параметров;
- валидация параметров (обязательные/опциональные, тип, диапазон);
- обработка глаголов HTTP;
- маршрутизация (функции/классы);

- выдача статических файлов (HTML, JS, CSS, изображения);
- выдача динамических данных (базы данных, сервисы);
- возврат значений и статусов HTTP.

Опциональные возможности:

- создание шаблонов для бэкенда;
- соединение с базами данных, ORM;
- ограничение скорости;
- асинхронные задачи.

В следующих подразделах мы напишем пример, использующий два фреймворка (Bottle и Flask). Они являются *синхронными*. Далее поговорим об альтернативах, в частности о сайтах, работающих с базами данных. Вы можете найти подходящий фреймворк Python для любого сайта, который только представите.

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` (пример 18.3).

Пример 18.3. 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 `/` (главная страница) обрабатывается функцией `home()`. Запустите данный сценарий сервера с помощью следующей команды:

```
$ python bottle1.py
```

Обратившись 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 порах разработки и тестирования.

Теперь вместо создания текста главной 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!!!
```

Дайте `bottle` задание возвращать содержимое этого файла, когда запрашивается главная страница. Сохраните данный сценарий как `bottle2.py` (пример 18.4).

Пример 18.4. 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` (пример 18.5).

Пример 18.5. 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: все, что находится в URL после `/echo/`, присваивается строковому аргументу `thing`, который передается функции `echo`. Чтобы увидеть развитие событий, 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-либо еще. Вы проверяли работу этих примеров, вводя URL в браузер и глядя на отображаемые страницы. Вы также можете использовать клиентские библиотеки, такие как `requests`, чтобы они выполняли работу за вас. Сохраните этот код как `bottle_test.py` (пример 18.6).

Пример 18.6. 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 19 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, при вызове функции `run()` можно 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, то попробуйте Flask. Он был создан в 2010 году как первоапрельская шутка, но реакция энтузиастов вдохновила его автора, Армина Ронахера, сделать его настоящим фреймворком. Он назвал результат Flask («склянка»), обыгрывая название Bottle — «бутылка».

Flask почти так же прост в использовании, как и Bottle, но поддерживает множество расширений, которые могут оказаться полезными в профессиональной веб-разработке, например аутентификацию с помощью Facebook и интеграцию с ба-

зами данных. Это решение мне нравится больше других веб-фреймворков Python, поскольку в нем сбалансированы простота применения и богатый набор функций.

Пакет `flask` включает в себя библиотеку 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 нужно внести несколько изменений.

- Во Flask каталог по умолчанию для статических файлов называется `static`, и URL для таких файлов тоже начинается со `/static`. Мы изменяем папку на `'.'` (текущий каталог) 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` (пример 18.7).

Пример 18.7. 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 главную 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, то Flask возвращает

особую отформатированную страницу, содержащую полезные сведения о том, что и где пошло не так. Даже больше: вы можете вводить команды с целью увидеть значения переменных в программе сервера.

Не устанавливайте параметр debug = True на производственных веб-серверах. Таким образом потенциальные злоумышленники получат 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 в отличие от Bottle? Flask содержит `jinja2` — более широкую систему шаблонов. Рассмотрим небольшой пример одновременного применения `jinja2` и Flask.

Создайте каталог `templates` и файл `flask2.html` внутри него (пример 18.8).

Пример 18.8. flask2.html

```
<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`, передаваемого нами, и отрисовывает его как HTML (я опущу функцию `home()` для экономии места). Сохраните этот файл под именем `flask2.py` (пример 18.9).

Пример 18.9. 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` утверждает, 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 каталоге `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 код, показанный в примере 18.10, как `flask3a.py`):

Пример 18.10. 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`, как это показано в примере 18.11; сохраните файл как `flask3b.py`.

Пример 18.11. 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 такой 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 `&`*`ключ1=знач1`*`&`*`ключ2=знач2`*`&...`

Вы также можете использовать оператор словаря **, чтобы передать несколько аргументов в шаблон с помощью одного словаря (назовите файл `flask3c.py`), как показано в примере 18.12.

Пример 18.12. 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)
```

Элемент `**kwargs` действует как конструкция `thing=thing, place=place`. Используя этот словарь, можно сэкономить немного времени, если входных аргументов много.

У языка шаблонов `jinja2` гораздо больше возможностей. Если вы работали на PHP, то увидите много общего.

Django

`Django` (https://www.djangoproject.com/) — очень популярный веб-фреймворк, написанный на Python, 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 подразделе SQLAlchemy на с. 348), позволяющий создавать автоматические веб-страницы для типичных функций баз данных `CRUD` (создание, замена, обновление, удаление), которые я рассматривал в главе 16. Он также включает автоматические страницы

администрирования для этих веб-страниц, но они предназначены для внутреннего использования среди программистов, а не размещения на публичных веб-страницах. Вам не обязательно использовать ORM именно для `Django`, если больше нравится применять что-то другое, например SQLAlchemy или прямые запросы SQL.

Другие фреймворки

Вы можете сравнить фреймворки с помощью онлайн-таблицы, размещенной по адресу http://bit.ly/web-frames:

- `fastapi` (https://fastapi.tiangelo.com/) обрабатывает как синхронные (WSGI), так и асинхронные (ASGI) вызовы, использует подсказки типов и генерирует тестовые страницы. Очень хорошо задокументирован. Рекомендую;
- `web2py` (http://www.web2py.com/) работает примерно с тем же, с чем и `django`, только немного в другом стиле;
- `pyramid` (https://trypyramid.com/) появился из более раннего проекта `pylons`, с точки зрения функционала похож на `django`;
- `turbogears` (http://turbogears.org/) поддерживает ORM, множество баз данных и несколько языков шаблонов;
- `wheezy.web` (http://pythonhosted.org/wheezy.web) более молодой фреймворк, оптимизирован для повышения производительности. Недавние исследования показали, что он работает быстрее других (http://bit.ly/wheezyweb);
- `molten` (https://moltenframework.com/) также использует подсказки типов, но поддерживает только WSGI;
- `apistar` (https://docs.apistar.com/) похож на `fastapi`, но это скорее инструмент для валидации API, чем веб-фреймворк;
- `masonite` (https://docs.masoniteproject.com/) — аналог Ruby on Rails или Laravel, написанный на Python.

Фреймворки для работы с базами данных

Интернет и базы данных — это неразлучная парочка: где одно, там и другое. В реальных приложениях Python в какой-то момент вам может понадобиться предоставить веб-интерфейс (сайт и/или API) реляционной базе данных.

Вы можете создать собственный интерфейс с помощью:

- веб-фреймворка, такого как Bottle или Flask;
- пакет для работы с БД, наподобие db-api или SQLAlchemy;
- драйвер для базы данных вроде pymysql.

Вместо этого вы также можете использовать один из пакетов, объединяющих Интернет и базы данных:

- connexion (https://connexion.readthedocs.io/);
- datasette (https://datasette.readthedocs.io/);

- sandman2 (https://github.com/jeffknupp/sandman2);
- flask-restless (https://flask-restless.readthedocs.io/).

Или можете использовать фреймворк со встроенной поддержкой баз данных, такой как Django.

Кроме того, ничто не заставляет вас выбирать именно реляционную базу данных. Если схема ваших данных может значительно изменяться — столбцы явно различаются в разных строках, — то вам стоит выбрать базу данных, не имеющую схемы. Это могут быть базы данных NoSQL, которые мы рассматривали в главе 16. Однажды я работал над сайтом, который изначально хранил данные в базе данных NoSQL, затем переключался на одну реляционную базу данных, затем на другую, потом на другую базу данных NoSQL и, наконец, возвращался к одной из реляционных.

Веб-сервисы и автоматизация

Мы рассмотрели традиционные веб-клиенты и серверные приложения, потребляющие и генерирующие HTML-страницы. Интернет оказался 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 информативной ссылке Python[1].

Вы можете использовать этот модуль непосредственно. Данная программа загружает главную страницу сайта 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.

[1] Если вы по какой-то причине не видите ее, то посетите сайт xkcd (http://xkcd.com/353).

Модуль webview

Вместо вызова браузера, как это делает модуль `webbrowser`, модуль `webview` отображает страницу в собственном окне, используя интерфейс, нативный для вашей машины. Чтобы установить его в ОС Linux или macOS:

```
$ pip install pywebview[qt]
```

В Windows:

```
$ pip install pywebview[cef]
```

Обратитесь к инструкциям по установке (https://oreil.ly/NiYD7), если у вас возникли проблемы.

Рассмотрим пример, в котором я передал модулю официальный сайт правительства США, показывающий текущее время:

```
>>> import webview
>>> url = input("URL? ")
URL? http://time.gov
>>> webview.create_window(f"webview display of {url}", url)
```

На рис. 18.1 показан полученный мной результат.

Чтобы остановить программу, убейте окно отображения.

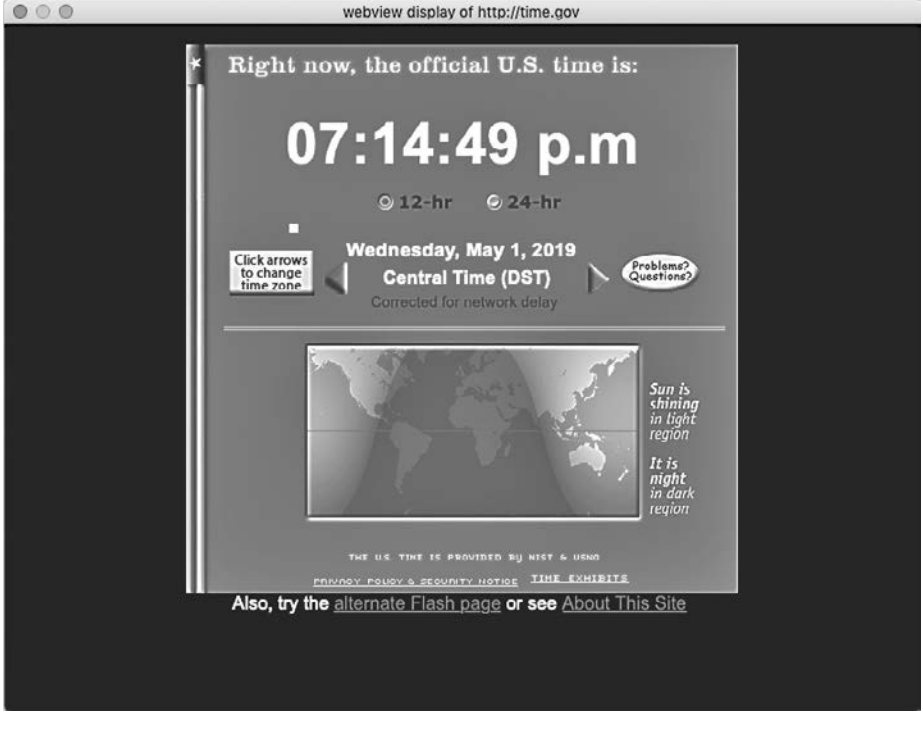

Рис. 18.1. Окно webview

REST API

Зачастую данные доступны только внутри веб-страниц. При желании 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, то 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, таких как JSON и XML (в главе 16 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.

Сервис *RESTful* использует *глаголы* HTTP определенными способами, описанными далее:

- `HEAD` получает информацию о ресурсе, но не его данные;
- `GET` получает данные ресурса с сервера. Это стандартный метод, используемый вашим браузером. `GET` не должен применяться для создания, изменения или удаления данных;
- `POST` создает новый ресурс;
- `PUT` заменяет существующий ресурс, создавая его, если его нет;
- `PATCH` частично обновляет 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.

Поиск и выборка данных

Иногда вам нужно получить немного больше информации — рейтинг фильма, цену акции или доступность продукта, — но требуемая информация доступна только на HTML-страницах, при этом 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*[1]. После того как содержимое было получено с одного из удаленных веб-серверов, *скрапер* анализирует ее, чтобы найти иголку в стоге сена.

Scrapy

Если вам нужно мощное решение, объединяющее в себе возможности краулера *и* скрапера, то стоит загрузить Scrapy (http://scrapy.org/):

```
$ pip install scrapy
```

Эта команда установит сам модуль и программу `scrapy`, работающую в командной строке.

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 *Scrapy at a Glance* (https://oreil.ly/8IYoe) и руководство (https://oreil.ly/4H_AW).

BeautifulSoup

Если у вас уже есть HTML-данные с сайта и вы просто хотите извлечь оттуда данные, то вам подойдет `BeautifulSoup` (https://oreil.ly/c43mV). 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, которые мы рассматривали в разделе «Текстовые строки: регулярные выражения» на с. 253, то довольно скоро столкнетесь с подобным беспорядком.

Чтобы установить `BeautifulSoup`, введите следующую команду (не забудьте поставить в конце `4`, 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. HTML-элемент `a` — это ссылка, а `href` — ее атрибут, который представляет собой место назначения ссылки. В примере 18.13 мы определим функцию `get_links()`, которая делает грязную работу, и основную программу, получающую один или несколько URL как аргументы командной строки.

Пример 18.13. links.py

```
def get_links(url):
    import requests
    from bs4 import BeautifulSoup as soup
    result = requests.get(url)
```

[1] Неприятный термин, если вы арахнофоб.

```
    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
```

Requests-HTML

Кеннет Ритц, автор популярного пакета для работы с веб-клиентами `requests`, написал новую библиотеку для скрапинга, которая называется Requests-HTML (http://html.python-requests.org/) (для Python 3.6 и более новых версий).

Она получает страницу и обрабатывает ее элементы, поэтому вы, например, можете найти все ссылки, все содержимое или атрибуты любого элемента HTML.

Библиотека имеет простой дизайн, похожий на `requests` и другие пакеты этого автора. В конечном счете она может быть проще в использовании, чем `BeautifulSoup` или Scrapy.

Давайте посмотрим фильм

Напишем полноценную программу.

Она будет искать видео с помощью API архива Internet Archive[1]. Это один из немногих API, которые позволяют получить анонимный доступ, *и* он будет существовать и после того, как данная книга будет напечатана.

---

[1] Если помните, я использовал еще один API для работы с архивами в основном примере главы 1.

Большинство веб-API требуют от вас получить ключ API и предоставлять его всякий раз, когда вы хотите получить к нему доступ. Почему? Это «трагедия общин» — бесплатные ресурсы с анонимным доступом зачастую используются слишком много и неправильно. Вот почему у нас не может быть хороших сервисов.

Программа, показанная в примере 18.14, делает следующее:

- приглашает вас ввести часть названия фильма или видео;
- ищет элемент с таким названием в Internet Archive;
- возвращает список идентификаторов, имен и описаний;
- перечисляет их и просит вас выбрать один;
- отображает это видео в вашем браузере;

Сохраните данный файл под именем `iamovies.py`.

Функция `search()` использует библиотеку `requests`, чтобы получить доступ к URL и результаты и преобразовать их в формат JSON. Другие функции обрабатывают все остальное. Вы увидите использование включений списков, разбиения строк и других возможностей, которые были представлены в предыдущих главах. (Номера строк не являются частью исходного кода; они будут использованы в упражнениях, чтобы найти нужные фрагменты кода.)

Пример 18.14. iamovies.py

```
""Найдем видео в Internet Archive
по фрагменту названия и отобразим его."""

import sys
import webbrowser
import requests

def search(title):
    """Возвращаем список кортежей, состоящих из трех элементов
       (идентификатор, заголовок, описание), которые описывают видеоролики,
       чьи заголовки частично соответствуют значению переменной title."""
    search_url = "https://archive.org/advancedsearch.php"
    params = {
        "q": "title:({}) AND mediatype:(movies)".format(title),
        "fl": "identifier,title,description",
        "output": "json",
        "rows": 10,
        "page": 1,
        }
    resp = requests.get(search_url, params=params)
    data = resp.json()
    docs = [(doc["identifier"], doc["title"], doc["description"])
            for doc in data["response"]["docs"]]
    return docs

def choose(docs):
    """Выведем номер строки, заголовок и частичное описание для
       каждого кортежа из списка :docs. Позволим пользователю выбрать
```

```
29        номер строки. Если он корректен, вернем первый элемент выбранного
30        кортежа ("идентификатор"). В противном случае вернем None."""
31     last = len(docs) - 1
32     for num, doc in enumerate(docs):
33         print(f"{num}: ({doc[1]}) {doc[2][:30]}...")
34     index = input(f"Which would you like to see (0 to {last})? ")
35     try:
36         return docs[int(index)][0]
37     except:
38         return None
39
40 def display(identifier):
41     """Отобразим видео из архива в браузере с помощью идентификатора """
42     details_url = "https://archive.org/details/{}".format(identifier)
43     print("Loading", details_url)
44     webbrowser.open(details_url)
45
46 def main(title):
47     """Найдем все фильмы, чье название соответствует значению переменной
48        :title. Узнаем выбор пользователя и отобразим его в браузере."""
49     identifiers = search(title)
50     if identifiers:
51         identifier = choose(identifiers)
52         if identifier:
53             display(identifier)
54         else:
55             print("Nothing selected")
56     else:
57         print("Nothing found for", title)
58
59 if __name__ == "__main__":
60     main(sys.argv[1])
```

Вот что я получил, запустив эту программу и выполнив поиск для буквосочетания `eegah`[1]:

```
$ python iamovies.py eegah
0: (Eegah) From IMDb : While driving thro...
1: (Eegah) This film has fallen into the ...
2: (Eegah) A caveman is discovered out in...
3: (Eegah (1962)) While driving through the dese...
4: (It's "Eegah" - Part 2) Wait till you see how this end...
5: (EEGAH trailer) The infamous modern-day cavema...
6: (It's "Eegah" - Part 1) Count Gore De Vol shares some ...
7: (Midnight Movie show: eegah) Arch Hall Jr...
Which would you like to see (0 to 7)? 2
Loading https://archive.org/details/Eegah
```

[1] В этом фильме Ричард Кил снялся в роли пещерного человека за много лет до того, как сыграл Челюсти в фильме о Джеймсе Бонде.

Он отобразил страницу в моем браузере, она готова к запуску (рис. 18.2).

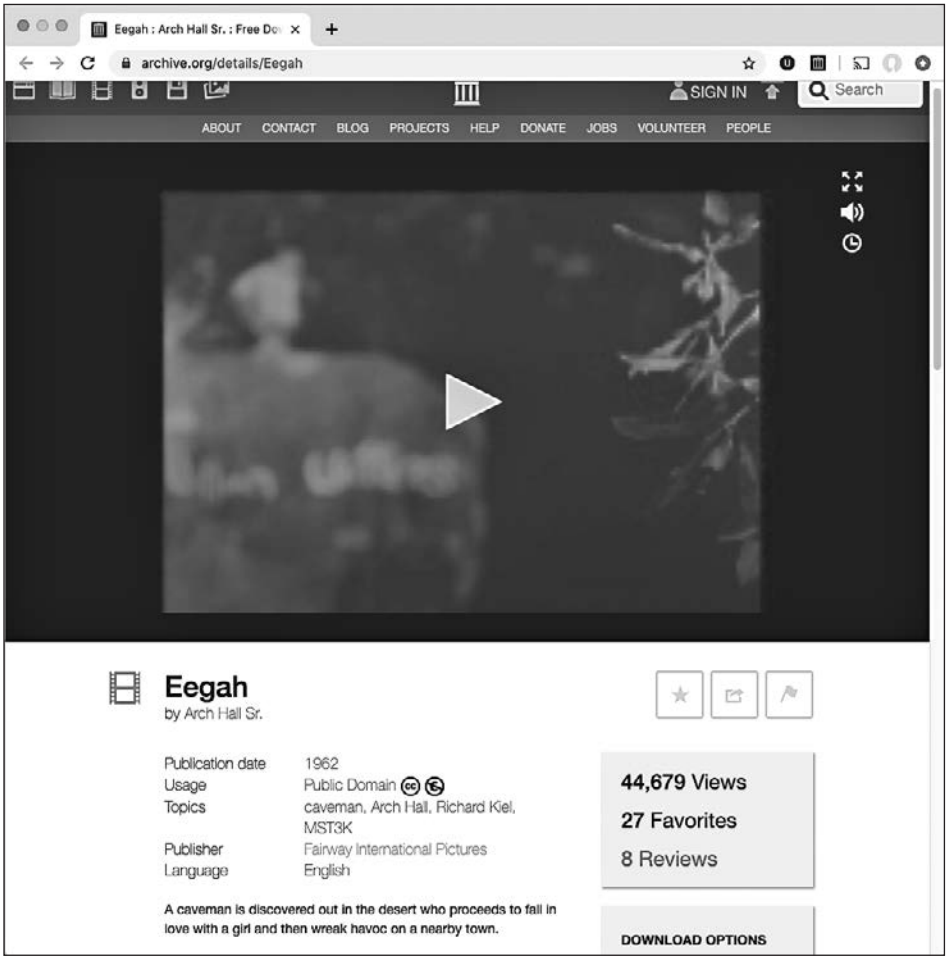

Рис. 18.2. Результат поиска фильма

Читайте далее

Следующая глава чрезвычайно практична, в ней рассматриваются основы современной разработки на Python. Узнайте же, как стать хладнокровным питонщиком.

Упражнения

18.1. Если вы еще не установили Flask, то сделайте сейчас. Это также позволит установить `werkzeug`, `jinja2` и, возможно, другие пакеты.

18.2. 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 этот порт для чего-то еще, то воспользуйтесь другим портом.

18.3. Добавьте функцию `home()` для обработки запросов к главной странице. Пусть она возвращает строку `It's alive!`.

18.4. 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>
```

18.5. Модифицируйте функцию `home()` вашего сервера, чтобы она использовала шаблон `home.html`. Передайте ей три параметра для команды `GET`: `thing`, `height` и `color`.

ГЛАВА 19
Быть питонщиком

> Всегда хотели отправиться во времени назад, чтобы сразиться с более молодой версией себя? Карьера в разработке ПО — то, что вам нужно!
>
> *Эллиот Лох*

Эта глава посвящена науке и искусству разработки с помощью Python, она содержит практические 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, 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. Вы должны написать программу так, чтобы получить правильные результаты, но вольны написать ее тем способом, который вам больше нравится. Такое соотношение использования левого и правого полушарий мозга необычно.

Начав свою карьеру в программировании, я также узнал, что в этой области существует множество ниш для разных задач и типов людей. Вы можете погрузиться в область компьютерной графики, операционных систем, бизнес-приложений и даже науки.

Если вы программист, то у вас мог быть похожий опыт. В противном случае можете попробовать начать программировать, чтобы посмотреть, подходит ли это вам или хотя бы помогает ли решить какие-либо задачи. Как я уже писал в данной

книге, знание математики здесь не так уж важно. Скорее всего, самое главное — способность 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... из тех источников, которые позволяют это делать.

Стандартная библиотека Python (http://docs.python.org/3/library) широка, глубока и довольно понятна.

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 улучшают что-то старое. 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 каталог пакетов Python (Python Package Index, PyPI) (https://pypi.org/). Ранее носивший имя *Cheese Shop* в честь скетча Monty Python, данный сайт постоянно обновляется — на момент написания этой книги он содержит более 113 000 пакетов. Когда вы используете `pip` (см. следующий раздел), он ищет пакет на сайте PyPI. Основная страница PyPI показывает самые свежие пакеты. Вы также можете выполнить прямой поиск, введя что-нибудь в строку поиска посередине главной страницы PyPI. Например, по ключевому слову `genealogy` определяется 21 совпадение, а по слову `movies` — 528.

Еще один популярный репозиторий — GitHub. Взгляните, какие пакеты Python популярны в данный момент (https://github.com/trending?l=python).

Сайт Popular Python recipes (http://bit.ly/popular-recipes) содержит более 4000 коротких программ Python на любую тему.

Установка пакетов

Существует множество способов установить пакет Python.

- Использовать `pip`, если есть такая возможность. На данный момент это самый распространенный метод установки. С помощью `pip` вы можете установить большинство пакетов.
- Задействовать `pipenv`, в нем объединены `pip` и `virtualenv`.
- Иногда вы можете применить менеджер пакетов своей операционной системы.
- Использовать `conda`, если вы выполняете большое количество научной работы и хотите задействовать дистрибутив Python, который называется Anaconda.

Прочтите раздел «Устанавливаем Anaconda» приложения Б на с. 536 для получения более подробной информации.

- С помощью исходного кода.

Если вам нужно установить несколько пакетов из одной области, то вы можете обнаружить дистрибутив, который уже содержит их. Например, глава 22 позволяет попробовать поработать с несколькими числовыми и научными программами, которые было бы трудно устанавливать вручную, но все они включены в дистрибутивы, такие как Anaconda.

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 работаете с более ранней версией Python и у вас не установлен `pip`, то можете скачать его, перейдя по ссылке http://www.pip-installer.org.

Простейший вариант использования `pip` — установка последней версии одного пакета с помощью следующей команды:

```
$ pip install flask
```

Вы увидите детали происходящего, просто чтобы не подумали, будто `pip` ленится: скачивание, запуск `setup.py`, установка файлов на диск и др.

Вы также можете дать `pip` команду установить определенную версию:

```
$ pip install flask==0.9.0
```

Или минимальную версию (это полезно, когда некая 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, то можете воспользоваться файлом требований (http://bit.ly/pip-require). Несмотря на обилие вариантов, простейший вариант использования — 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
```

Еще несколько примеров:

- установите последнюю версию: `pip install --upgrade` *`пакет`*;
- удалите пакет: `pip uninstall` *`пакет`*.

virtualenv

Стандартный способ установки сторонних пакетов — это использование `pip` и `virtualenv`. Я покажу, как устанавливать `virtualenv`, в разделе «Установка virtualenv» приложения Б на с. 535.

Виртуальное окружение — это обычный каталог, содержащий интерпретатор Python, несколько других программ наподобие `pip`, а также некие пакеты. Вы можете *активизировать* его, запустив сценарий оболочки, который лежит в каталоге `bin` этого виртуального окружения. Это изменит значение переменной окружения `$PATH`, и теперь ваша оболочка будет знать, где искать программы. Активизируя виртуальное окружение, вы помещаете его каталог `bin` перед остальными каталогами в переменной `$PATH`. В результате, когда вы введете такие команды, как `pip` или `python`, ваша оболочка сначала найдет те программы, которые располагаются в вашем виртуальном окружении, а не те, что находятся в системных каталогах наподобие `/bin`, `/usr/bin` или `/usr/local/bin`.

Вы не захотите устанавливать программы в эти системные каталоги в любом случае, поскольку:

- у вас нет разрешения на запись в них;
- даже если бы вы могли выполнить запись, переписывание стандартных программ вашей системы (такой как `python`) может вызвать проблемы.

pipenv

Недавно появился пакет `pipenv` (https://github.com/pypa/pipenv), в котором объединены уже знакомые нам `pip` и `virtualenv`. Вдобавок он решает проблемы с зависимостями, которые могут возникнуть при использовании `pip` в разных окружениях (например, на локальной машине, а также на этапе подготовки и развертывания):

```
$ pip install pipenv
```

Использовать этот пакет рекомендует Python Packaging Authority (https://www.pypa.io/) — рабочая группа, которая старается улучшить процесс взаимодействия с пакетами в Python. Эта группа не занимается разработкой ядра Python, поэтому `pipenv` не входит в состав стандартной библиотеки.

Менеджер пакетов

Операционная система macOS от Apple 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. Самые популярные — `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 пакет еще совсем новый или же автор просто не сделал его доступным через `pip`. Чтобы создать пакет, вы, как правило, делаете следующее:

- загружаете код;
- извлекаете файлы с помощью `zip`, `tar` или другого подходящего инструмента, если они заархивированы или сжаты;
- запускаете команду `python setup.py install` в каталоге, который содержит файл `setup.py`.

Как всегда, вам следует быть осторожными с тем, что вы скачиваете 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 данной 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 такие инструменты, как текстовые редакторы, отладчики, поиск 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. 19.1 показан ее начальный экран.

Рис. 19.1. Начальный экран PyCharm

IPython

IPython (http://ipython.org/) начал свой путь как улучшенная терминальная (текстовая) IDE для Python, но затем эволюционировал и приобрел графический интерфейс, похожий на *блокнот*. Он интегрировал множество пакетов, рассмотренных в этой книге, включая Matplotlib и NumPy, и стал популярным инструментом для научных вычислений.

Вы можете установить базовую текстовую версию (как уже догадались) с помощью команды `pip install ipython`. Запустив ее, вы увидите нечто похожее на следующий блок:

```
$ ipython
Python 3.7.3 (v3.7.3:ef4ec6ed12, Mar 25 2019, 16:39:00)
Type 'copyright', 'credits' or 'license' for more information
IPython 7.3.0 -- An enhanced Interactive Python. Type '?' for help.

In [1]:
```

Как вам известно, стандартный интерпретатор Python использует приглашения `>>>` и `...`, чтобы указать, где и когда вам нужно вводить код. IPython отслеживает все, что вы вводите, в списке, который называется `In`, а все выходные данные — в списке `Out`. Каждый элемент списка `In` может быть длиннее одной строки, поэтому отправлять свои данные следует, удерживая нажатой клавишу Shift и нажимая клавишу Enter.

Рассмотрим однострочный пример:

```
In [1]: print("Hello? World?")
Hello? World?

In [2]:
```

Элементы в списках `In` и `Out` нумеруются автоматически, позволяя получить доступ ко всем входным и выходным данным.

Если после имени переменной вы введете знак `?`, то IPython подскажет вам ее тип, значение, способы создания переменной подобного типа, а также некоторые пояснения:

```
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 возможность такой IDE, как IPython. Если вы нажмете клавишу Tab после того, как введете некие символы, то IPython покажет все переменные, ключевые слова и функции, начинающиеся с этих символов. Определим некоторые переменные, 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`, а затем нажмете клавишу Tab, то увидите только переменную `fee`, единственную переменную в этой программе, которая начинается с буквосочетания `fe`:

```
In [11]: fee
Out[11]: 1
```

У IPython много других возможностей. Взгляните на это руководство (https://oreil.ly/PIvVK), чтобы получить представление о них.

Jupyter Notebook

Jupyter (https://jupyter.org/) — это эволюционировавший IPython. В его названии объединены языки Julia, Python и R — все они популярны в области науки о данных и научных вычислений. Jupyter Notebooks — современный способ разработки и публикации вашего кода, содержащий документацию к этим языкам.

Если вы хотите исследовать его, не устанавливая на свой компьютер, то можете запустить его (https://jupyter.org/try) в браузере.

Чтобы установить Jupyter Notebook локально, введите команду `pip install jupyter`. Запустите его с помощью `jupyter notebook`.

JupyterLab

JupyterLab — это Jupyter Notebook нового поколения, в конечном счете он заменит старую версию. Как и в случае с Notebook, вы сначала можете попробовать (https://jupyter.org/try) JupyterLab в своем браузере. Установить его локально можно с помощью команды `pip install jupyterlab`, а запустить — с помощью `jupyter lab`.

Именование и документирование

Вы не вспомните написанное. 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. Это могут быть комментарии; 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, то вам следует назвать переменные так, чтобы было понятно их функционирование, а не производить на свет кучу волшебного кода. Небольшой тест также не повредит (пример 19.1):

Пример 19.1. ftoc1.py

```
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()`.

В примере 19.2 показан переделанный код.

Пример 19.2. ftoc2.py

```
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))
```

Добавление подсказок типов

Статические языки требуют от вас определять типы переменных, они могут отлавливать ошибки на этапе компиляции. Как вам известно, Python так не делает, вы можете столкнуться с багом только при запуске кода. Переменные в Python — это имена, они лишь ссылаются на реальные объекты. Объекты имеют строго определенный тип, но имя может указывать на разные объекты в различные моменты времени.

В реальном коде (на Python и других языках) имя, как правило, ссылается на конкретный объект. Было бы полезно, по крайней мере в документации, если бы мы могли оставлять аннотации (к объектам, возвращаемым значениям функций и т. д.), указывавшие бы на ожидаемый тип объекта, на который они ссылаются. В таком случае разработчикам не пришлось бы просматривать большое количество кода, чтобы увидеть, как переменная должна вести себя.

В Python версий 3.x для решения этой проблемы были добавлены *подсказки типов* (или *аннотации типов*). Ими пользоваться не обязательно, они не задают жесткий тип переменной. Подсказки типов предназначены для того, чтобы помочь разработчикам, привыкшим к статическим языкам, понять, где *должны* быть объявлены типы переменных.

Подсказка для функции, которая преобразует строку в число, может выглядеть так:

```
def num_to_str(num: int) -> str:
    return str(num)
```

Это лишь подсказки, они не меняют способ работы Python. Они нужны для документирования, однако люди придумывают и другое применение. Например, веб-фреймворк FastAPI (https://fastapi.tiangolo.com/) использует подсказки для того, чтобы сгенерировать веб-документацию, а также лайв-формы для тестирования.

Тестирование кода

Возможно, вы уже знаете, однако на всякий случай напомню: даже тривиальные изменения в коде могут сломать вашу программу. В Python недостает проверки типов, присущей статическим языкам, что упрощает некоторые аспекты программирования, но повышает вероятность получить нежелательные результаты. Тестирование — это важно.

Самый простой способ протестировать программы, написанные на Python, — добавить операторы `print()`. Read-Evaluate-Print Loop (REPL) интерактивного интерпретатора позволяет быстро изменять код и тестировать изменения. Однако в производственном коде операторы `print()` использовать не стоит, поэтому вам нужно помнить о том, что их все следует удалять.

Программы pylint, pyflakes, flake8 или PEP-8

Следующий шаг перед созданием настоящих программ для тестирования — 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 клетку). В примере 19.3 показана практически бессмысленная программа, в которой есть логическая и стилистическая ошибки.

Пример 19.3. style1.py

```
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`, то 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`, начинающаяся с `E`, указывает на `Error`, которая заключается в том, что мы не присвоили значение переменной до ее вывода на экран. Взгляните на пример 19.4, чтобы увидеть, как можно это исправить.

Пример 19.4. style2.py

```
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
```

Контролер `pylint` хочет увидеть строку документации (короткий текстовый фрагмент в верхней части модуля или функции, описывающий код) и считает, что короткие имена переменных, такие как `a`, `b` и `c`, не очень аккуратны. Сделаем `pylint` более счастливым, а наш код — еще лучше (пример 19.5):

Пример 19.5. style3.py

```
"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 один контролер стиля — `pep8` (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.

Пакет unittest

Мы убедились, что больше не оскорбляем чувство стиля богов кода, поэтому теперь можно перейти к настоящим тестам логики вашей программы.

Хорошим тоном являются написание и запуск тестовых программ до отправки кода в систему контроля исходного кода. Написание тестов поначалу может быть утомительным, но они действительно помогают находить проблемы быстрее, особенно *регрессионные тесты* (суть которых заключается в том, чтобы сломать то, что раньше работало). Болезненный опыт учит всех разработчиков: 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://oreil.ly/ImFmE). 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` (пример 19.6).

Пример 19.6. 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 введенный текст 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`, скажем метода `assertEqual`, показанного в примере 19.7.

Сохраните этот сценарий тестирования под именем `test_cap.py`.

Пример 19.7. 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()`. Их задача — выделение и освобождение внешних ресурсов, необходимых для тестов, таких как соединение с базой данных или создание неких тестовых данных. В нашем случае тесты автономны, и нам даже не нужно определять методы `setUp()` и `tearDown()`, однако создать их пустые версии не повредит. Сердце наших тестов — 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://oreil.ly/x1IV8) 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://oreil.ly/CNKNl). Изменим файл `cap.py` так, чтобы в нем вместо функции `capitalize()` использовалась функция `title()` (пример 19.8).

Пример 19.8. Возвращаемся к cap.py

```
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 (пример 19.9).

Пример 19.9. Возвращаемся к test_cap.py

```
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 текст документации 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` (пример 19.10).

Пример 19.10. Снова возвращаемся к 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` (пример 19.11).

Пример 19.11. Снова возвращаемся к 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 пограничные случаи, но разработчики часто забывают о подобных ситуациях, когда речь идет об их собственном коде.

Пакет `unittest` предоставляет небольшой, но мощный набор операторов, позволяющих проверять значения, убеждаться в наличии необходимого класса, определять, сгенерировалась ли ошибка, и т. д.

Пакет doctest

Второй пакет для тестирования стандартной библиотеки — `doctest` (http://bit.ly/py-doctest). Он позволяет писать тесты внутри строки документации, которые и сами будут служить документацией. Он выглядит как интерактивный интерпретатор: символы `>>>`, за ними следует вызов, а затем результаты в следующей строке. Вы можете запустить некоторые тесты в интерактивном интерпретаторе и просто вставить результат в свой тестовый файл. Модифицируем файл `cap.py` (убрав тот проблемный тест с кавычками) и сохраним его под именем `cap2.py`, как показано в примере 19.12.

Пример 19.12. cap2.py

```
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 cap2.py
```

Запустите его с опцией `-v`, чтобы увидеть произошедшее на самом деле:

```
$ python cap2.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://oreil.ly/gWK6r) — 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 наш последний тестировщик `Unittest` и сохраним его под именем `test_cap_nose.py` (пример 19.13).

Пример 19.13. test_cap_nose.py

```
import cap2
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 какой-то причине людям нравится писать тестовые фреймворки для Python. Если вам любопытно, то можете взглянуть на другие популярные решения:

- `tox` (http://tox.readthedocs.org/);
- `py.test` (https://pytest.org/);
- `green` (https://github.com/CleanCut/green).

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 при возникновении нового кода. Таким образом, каждый будет знать, *сломал ли кто-то сборку* и убежал обедать пораньше — или на новую работу.

Эти системы велики, и я не буду рассматривать детали их установки и использования. Если они вам когда-нибудь понадобятся, то 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, в данный момент выглядит наиболее предпочтительным инструментом для постоянной интеграции;
- `travis-ci` (http://travis-ci.com/) — эта система автоматизирует проекты, размещенные на GitHub, бесплатна для проектов с открытым исходным кодом;
- `circleci` (https://circleci.com/) — эта система коммерческая, но бесплатна для частных проектов и проектов с открытым исходным кодом.

Отладка кода

> Отладка — почти как расследование преступления, где главный преступник — тоже вы.
>
> *Филипе Фортес*

> Все знают: отладка кода вдвое сложнее, чем его написание. Так что если вы пишете код настолько умно, насколько можете, то по определению недостаточно сообразительны, чтобы его отлаживать.
>
> *Брайан Керниган*

Всегда тестируйте код. Чем лучше ваши тесты, тем меньше кода вам потом придется исправлять. Но иногда баги все равно случаются, и их нужно исправлять при обнаружении.

Обычно код ломается из-за действий, которые вы сделали только что. Поэтому обычно отладка выполняется «снизу вверх», начиная с ваших самых последних изменений[1].

Но иногда причина кроется где-то еще, в чем-то, чему вы доверяли и думали, что оно работает. Вы могли бы подумать: если бы проблема была в том, чем пользовались многие люди, то кто-то уже заметил бы ее. Но так бывает не всегда. Самые хитрые баги, которые требовали на свое исправление больше недели, появились из внешних источников. Поэтому после того, как обвините человека в зеркале, пересмотрите свои предположения. Данный подход называется нисходящим и занимает больше времени.

Далее мы рассмотрим некоторые приемы отладки, начиная с быстрых и «грязных» и заканчивая медленными и зачастую такими же «грязными».

Функция print()

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': {}}
```

[1] Вы как детектив: «Я знаю, что я здесь! И если не выйду с поднятыми руками, то приду за собой!»

Зачастую на экран следует выводить `locals()` и `globals()`.

Если в вашем коде также встречается и стандартный вывод данных, то можете написать собственные сообщения об ошибке в поток вывода для ошибок с помощью конструкции `print(stuff, file=sys.stderr)`.

Отладка с помощью декораторов

Как вы уже знаете из раздела «Декораторы» главы 9, декоратор может вызывать код, располагающийся до или после функции, не модифицируя его внутри нее самой. Это значит, что вы можете использовать декоратор для выполнения какого-либо действия до или после вызова любой функции, а не только тех, которые написали вы. Определим декоратор `dump`, позволяющий вывести на экран входные аргументы и выводимые значения любой функции по мере ее вызова (разработчики знают, что выходные данные нужно декорировать), как показано в примере 19.14.

Пример 19.14. dump.py

```
def dump(func):
    "Print input arguments and output value(s)"
    def wrapped(*args, **kwargs):
        print("Function name:", func.__name__)
        print("Input arguments:", ' '.join(map(str, args)))
        print("Input keyword arguments:", 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 (пример 19.15).

Пример 19.15. test_dump.py

```
from dump 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([('first', 100), ('next', 98.6),
    ('last', -40)])
Output: ([6, 10], {'first': 200, 'next': 197.2, 'last': -80})
```

Отладчик pdb

Эти приемы полезны, но иногда ничто не сможет заменить настоящий отладчик. Большинство IDE содержат отладчики, возможности и пользовательские интерфейсы которых могут варьироваться. В данном подразделе я опишу использование стандартного отладчика Python pdb (https://oreil.ly/IIN4y).

Если вы запускаете программу с флагом -i, то 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, которая возникла в ранние дни компьютеризации и 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»*. Прописные буквы в них могут быть расставлены неправильно, так что нам нужно исправить это при выводе на экран. В файле также могут быть лишние пробелы, следует 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). В примере 19.16 показан файл с данными.

Пример 19.16. cities.csv

```
France, Paris
venuzuela,caracas
  LithuaniA,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, когда приходит время:

```
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 таково требование 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 (пример 19.17).

Пример 19.17. capitals.py

```
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 cities.csv
Paris,France
Caracas,Venuzuela
Vilnius,Lithuania
```

Выглядит отлично! Программа прошла один тест, поэтому отправим ее в производство, обрабатывать столицы и страны со всего мира, пока она не ошибется на этом файле данных (пример 19.18).

Пример 19.18. cities2.csv

```
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
```

Несмотря на то что в файле было 15 строк, программа завершается после вывода всего пяти:

```
$ 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 местах потенциального возникновения ошибки, но посмотрим, сможет ли помочь отладчик.

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»), то 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 (которые бы выдали сообщение об ошибке).

Введите `s` (*step* — «шаг»), чтобы пройти по отдельным строкам кода. Это позволит пройти по *всем* строкам кода Python: вашим, стандартной библиотеки и любых других применяемых вами модулей. Используя 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 не уверены в том, где конкретно есть проблема, а `n` — будучи уверенными, что некая функция не вызывает проблем,

особенно когда она длинная. Зачастую вы будете проходить построчно весь свой код и пропускать библиотечный, поскольку подразумевается, что он хорошо протестирован. Используем `s` с целью начать двигаться от начала программы к функции `process_cities()`:

```
(Pdb) s

> /Users/williamlubanovic/book/capitals.py(12)<module>()
-> if __name__ == '__main__':</pre>

(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 используем одну из главных особенностей отладчика — *точки останова*. Такая точка 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 выглядит невинно. Единственное место, где мы можем вернуться из функции до прочтения всех данных, — это строка 6 (`return`). Поставим точку останова на этой строке:

```
(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` с целью увидеть только что прочитанное нами:

```
(Pdb) p line

'ecuador,quito'
```

Серьезно? Столица называется `quito`? Ваш 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, то 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 строки кода, текущую строку (`->`) и все имеющиеся точки останова (`B`). Вызов команды `l` без аргументов выведет все строки, начиная с точки предыдущего вызова этой команды, 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, как показано в примере 19.19, чтобы выполнялась проверка на полное совпадение со строкой `quit` без всяких других символов.

Пример 19.19. capitals.py

```
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
```

Это был краткий обзор отладчика — его достаточно для демонстрации того, что вы можете сделать и какие команды будете использовать бóльшую часть времени.

Помните: больше тестов — меньше отладки.

Функция breakpoint()

В Python 3.7 появилась новая встроенная функция `breakpoint()`. Если вы добавите ее в свой код, то отладчик запустится автоматически и будет останавливаться в каждой локации. Без нее вам придется запускать такой отладчик, как `pdb`, и расставлять точки останова вручную, как вы уже видели ранее.

Отладчик, используемый по умолчанию, вам уже знаком (`pdb`), но его можно изменить, установив переменную окружения `PYTHONBREAKPOINT`. Например, вы можете использовать удаленный отладчик, работающий на основе веб-подключения `web-pdb` (https://pypi.org/project/web-pdb):

```
$ export PYTHONBREAKPOINT='web_pdb.set_trace'
```

Официальная документация несколько суховата, но хорошие обзоры этой функции можно найти здесь (https://oreil.ly/9Q9MZ) и здесь (https://oreil.ly/2LJKy).

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, например временную метку или имя пользователя, запустившего программу. Зачастую журналы ежедневно *ротируются* (переименовываются) и сжимаются, благодаря чему не переполняют ваш диск и не создают проблем. Если ваша программа работает «не так», то вы можете просмотреть соответствующий файл журнала и увидеть, что произошло. Содержимое исключений особенно полезно записывать в журнал, поскольку оно подсказывает 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 они будут казаться осмысленными, но поначалу 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-либо еще;
- средства *форматирования* выходных данных;
- *фильтры*, принимающие 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 оператор `print()`, имеющий несколько личностей.

Но это полезно. Вы можете выполнить поиск определенного значения уровня в журнале с целью найти конкретные сообщения, сравнить временны́е метки и увидеть, что случилось, прежде чем упал ваш сервер, и т. д.

Глубокое погружение в документацию отвечает 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, то они разделяют уровни иерархии таких объектов, каждый из которых потенциально имеет разные приоритеты. Это значит, что объект с именем `quark` выше объекта `quark.charmed`. На вершине иерархии находится *корневой объект журналирования* с именем `''`.

До сего момента мы только выводили сообщения, и это практически не отличается от функции `print()`. Чтобы направить сообщения в разные места назначения, используем *обработчики*. Самое распространенное место — *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 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, такие как электронная почта, веб-серверы, экраны и файлы.

Наконец, вы можете управлять *форматом* сообщений журнала. В нашем первом примере использовался формат 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 собственные переменные.

Пакет `logging` содержит гораздо больше особенностей, чем можно описать в приведенном небольшом обзоре. Вы можете писать в несколько журналов одновременно, указывая разные приоритеты и форматы. Данный пакет довольно гибок, но иногда это достигается за счет простоты.

Оптимизация кода

Обычно Python довольно быстр, 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. 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, что-то сделать, получить новое время и вычесть из него первое. Напишем соответствующий код, показанный в примере 19.20, и назовем файл (как бы вы думали?) `time1.py`.

Пример 19.20. time1.py

```
from time import time

t1 = time()
num = 5
num *= 2
print(time() - t1)
```

Здесь мы измеряем время, которое требуется на присвоение значения `5` переменной `num` и умножение его на `2`. Данный пример *не* является реалистичным тестом производительности, это лишь образец того, как замерить время выполнения произвольного кода. Попробуйте запустить его несколько раз, чтобы увидеть — 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 2–3 миллионные доли секунды. Попробуем выполнить что-то помедленнее, скажем функцию `sleep()`[1]. Если мы усыпим выполнение на секунду, то наш таймер покажет значение чуть больше секунды. В примере 19.21 показан код, сохраните его в файле под именем `time2.py`.

Пример 19.21. time2.py

```
from time import time, sleep

t1 = time()
sleep(1.0)
print(time() - t1)
```

[1] Во многих книгах в примерах, где требуется пропустить какое-то время, приводится расчет чисел Фибоначчи, но я бы лучше проспал это время.

Чтобы быть уверенными 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 стать стыдно.

Существует более удобный способ измерить время выполнения таких фрагментов кода, как этот, — использование стандартного модуля `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 следующим образом: `timeit.timeit(`*`код, число, количество_раз`*`)`.

В примерах этого подраздела код должен находиться в кавычках, чтобы он выполнялся не после нажатия клавиши Return, а лишь внутри функции `timeit()`. (В следующем подразделе вы увидите, как можно измерить время выполнения некой функции, передав ее имя в функцию `timeit()`.) Запустим предыдущий пример и измерим время его выполнения. Назовем этот файл `timeit1.py` (пример 19.22).

Пример 19.22. 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
```

Эти две строки кода снова 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` (пример 19.23).

Пример 19.23. timeit2.py

```
from timeit import repeat
print(repeat('num = 5; num *= 2', number=1, repeat=3))
```

Попробуйте запустить его, чтобы увидеть дальнейшее развитие событий:

```
$ 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? Так могло произойти по многим причинам. Например, 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.

Использование `timeit()` означает, что вы оборачиваете код, производительность которого измеряете, в строку. А если у вас несколько строк? Вы могли бы передать их как строку с тройными кавычками, но такую нотацию будет сложно прочитать.

Определим ленивую функцию `snooze()`, которая дремлет одну секунду, как и мы все время от времени.

Для начала обернем саму функцию `snooze()`. Нам нужно включить аргументы `globals=globals()` (это поможет Python найти функцию `snooze`) и `number=1` (запустить ее только один раз; по умолчанию используется значение `1000000`, а у нас нет столько времени):

```
>>> import time
>>> from timeit import timeit
>>>
>>> def snooze():
...     time.sleep(1)
...
>>> seconds = timeit('snooze()', globals=globals(), number=1)
>>> print("%.4f" % seconds)
1.0035
```

Или мы можем использовать декоратор:

```
>>> import time
>>>
>>> def snooze():
...     time.sleep(1)
...
>>> def time_decorator(func):
...     def inner(*args, **kwargs):
...         t1 = time.time()
...         result = func(*args, **kwargs)
...         t2 = time.time()
...         print(f"{(t2-t1):.4f}")
...         return result
...     return inner
...
>>> @time_decorator
... def naptime():
...     snooze()
...
>>> naptime()
1.0015
```

Еще один вариант — задействовать менеджер контекста:

```
>>> import time
>>>
>>> def snooze():
```

```
...     time.sleep(1)
...
>>> class TimeContextManager:
...     def __enter__(self):
...         self.t1 = time.time()
...         return self
...     def __exit__(self, type, value, traceback):
...         t2 = time.time()
...         print(f"{(t2-self.t1):.4f}")
...
>>>
>>> with TimeContextManager():
...     snooze()
...
1.0019
```

Метод `__exit()__` принимает три дополнительных аргумента, которые мы здесь не использовали; мы могли бы применить `*args` вместо них.

О'кей, мы узнали множество способов измерить время, за которое выполняется код. Теперь 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 способ 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, читабельность и стиль. Перед вами файл `time_lists.py` (пример 19.24).

Пример 19.24. 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: в этом тесте мы вызываем функцию `timeit()`, передавая ей имя функции в качестве первого аргумента вместо кода. Запустим ее:

```
$ 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 ускорить свой код.

Cython, NumPy и расширения C

Если вы усердно работаете, но все еще не можете достичь необходимой производительности, то у вас есть и другие варианты.

Cython (http://cython.org/) — 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 значительно ускорят научные вычисления, выполняющиеся в циклах, — в 1000 раз. Документацию и примеры см. в Cython wiki (https://oreil.ly/MmW_v).

В главе 22 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», то 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.

У языка Python нет владельца, поэтому никто так сильно не старается сделать его быстрее. Возможно, вы 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 (называемый *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 более чем в шесть раз и до 20 раз в отдельных случаях. PyPy работает с Python 2 и 3. Вы можете скачать его и использовать вместо CPython. PyPy постоянно улучшается и однажды может заменить CPython. Чтобы узнать, подходит ли он вам, посетите его официальный сайт.

Numba

Вы можете использовать пакет Numba (http://numba.pydata.org/), чтобы динамически скомпилировать свой код в машинный и ускорить его.

Устанавливается этот пакет так же, как и другие:

```
$ pip install numba
```

Сначала измерим время выполнения функции, которая высчитывает гипотенузу:

```
>>> import math
>>> from timeit import timeit
>>> from numba import jit
>>>
>>> def hypot(a, b):
...     return math.sqrt(a**2 + b**2)
...
>>> timeit('hypot(5, 6)', globals=globals())
0.6349189280000189
>>> timeit('hypot(5, 6)', globals=globals())
0.6348589239999853
```

Используйте декоратор `@jit`, чтобы ускорить все вызовы после первого:

```
>>> @jit
... def hypot_jit(a, b):
...     return math.sqrt(a**2 + b**2)
...
>>> timeit('hypot_jit(5, 6)', globals=globals())
0.5396156099999985
>>> timeit('hypot_jit(5, 6)', globals=globals())
0.1534771130000081
```

Примените декоратор `@jit(nopython=True)`, чтобы избежать накладных расходов, связанных с использованием обычного интерпретатора Python:

```
>>> @jit(nopython=True)
... def hypot_jit_nopy(a, b):
...     return math.sqrt(a**2 + b**2)
...
>>> timeit('hypot_jit_nopy(5, 6)', globals=globals())
```

```
0.18343535700000757
>>> timeit('hypot_jit_nopy(5, 6)', globals=globals())
0.15387067300002855
```

Пакет Numba особенно полезен в связке с NumPy и другими пакетами, выполняющими вычисления.

Управление исходным кодом

Работая над небольшой группой программ, вы обычно можете отслеживать изменения, внесенные собственноручно, — 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, то управление исходным кодом становится необходимостью. Для этой области было создано множество коммерческих и бесплатных решений. Наиболее популярные в мире открытого исходного кода (где и живет Python) — Mercurial и Git. Они оба являются примерами *распределенных* систем контроля версий, которые создают несколько копий репозиториев кода. Ранние системы наподобие Subversion работают на одном сервере.

Mercurial

Mercurial (http://mercurial-scm.org/) написан на Python. Научиться пользоваться им довольно легко, он имеет множество подкоманд для скачивания кода из репозитория Mercurial, добавления файлов, проверки на наличие изменений и объединения изменений из разных источников. Сайт bitbucket (https://bitbucket.org/) и другие предлагают бесплатный или коммерческий хостинг.

Git

Изначально Git (http://git-scm.com/) 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://oreil.ly/U2Rmy). Если у вас установлена программа `git`, то вы можете скачать примеры с помощью следующей команды:

```
$ git clone https://github.com/madscheme/introducing-python
```

Вы также можете скачать код, нажав 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 поработать с ним, то прочтите инструкцию по установке (http://bit.ly/git-install). Здесь я буду говорить о версии с командной строкой, но вам могут быть интересны сайты наподобие GitHub, предоставляющие дополнительные услуги, которые в ряде случаев 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 новый каталог и перейдем в него:

```
$ mkdir newdir
$ cd newdir
```

Создадим локальный репозиторий `git` в текущем каталоге newdir:

```
$ git init

Initialized empty Git repository in /Users/williamlubanovic/newdir/.git/
```

Создадим в каталоге newdir файл с кодом, который называется `test.py` (показан в примере 19.25), содержащий следующее.

Пример 19.25. test.py

```
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 данное обстоятельство:

```
$ 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"` — ваш *комментарий*. Если вы опустите ее, то `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 потере его оригинала. Внесем изменение в файл `test.py` — заменим `Oops` на `Ops!` и сохраним файл (пример 19.26).

Пример 19.26. Возвращаемся к test.py

```
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, то `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; в примерном переводе она 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 с целью гарантировать отправку всех изменений. Теперь можно отправить изменения:

```
$ git commit -m "my first change"

 [master e1e11ec] my first change
  1 file changed, 1 insertion(&plus;), 1 deletion(-)
```

Если вы хотите увидеть все те ужасные вещи, которые проделывали с файлом `test.py`, начиная с недавних, то 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
```

Распространение ваших программ

Вы знаете, что ваши файлы нужно установить в файлы и каталоги, а также то, что программу, написанную на Python, можно запустить с помощью интерпретатора.

Менее известен тот факт, что интерпретатор Python тоже может выполнять код, размещенный в архивах ZIP. Еще менее известно то, что особые архивы, которые называются pex-файлами (https://pex.readthedocs.io/), также могут быть скомпилированы.

Клонируйте эту книгу

Вы можете получить копию всех программ из данной книги. Посетите репозиторий Git (https://oreil.ly/FbFAE) и следуйте инструкциям по их копированию на ваш локальный компьютер. Если у вас есть `git`, то 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 ненужном вам и слишком мало о том, что вам интересно. Позвольте мне порекомендовать ресурсы, связанные с Python, которые я считаю полезными.

Книги

Я обнаружил, что книги, представленные в следующем списке, особенно полезны. Их уровень варьируется от начального до продвинутого, в них описываются и Python 2, и Python 3.

- *Бизли Д., Джонс Б.* Python. Книга рецептов. — М.: ДМК-Пресс, 2020.
- *Рейтц К., Шлюссер Т.* Автостопом по Python. — СПб.: Питер, 2017.
- *Barry P.* Head First Python (2nd Edition). — O'Reilly, 2016.
- *Beazley D. M.* Python Essential Reference. 5th ed. — Addison-Wesley, 2019.
- *Gorelick M., Ozsvald I.* High Performance Python. — O'Reilly, 2014.
- *Maxwell A.* Powerful Python. — Powerful Python Press, 2017.
- *McKinney W.* Python for Data Analysis: Data Wrangling with Pandas, NumPy and IPython. — O'Reilly, 2012.
- *Ramalho L.* Fluent Python. — O'Reilly, 2015.
- *Slatkin B.* Effective Python. — Addison-Wesley, 2015.
- *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:

- Python for You and Me (https://pymbook.readthedocs.io/) — введение в Python, в котором также рассматриваются нюансы работы с ОС Windows;
- Real Python (http://realpython.com/) — многие авторы внесли свой вклад в создание этих руководств;
- Learn Python the Hard Way (http://learnpythonthehardway.org/book), автор Зед Шоу;
- Dive Into Python 3 (https://oreil.ly/UJcGM), автор Марк Пилгрим;
- Mouse Vs. Python (http://www.blog.pythonlibrary.org/), автор Майкл Дрисколл.

Если вам интересно узнавать о том, что происходит в мире Python, то обратите внимание на эти новостные сайты:

- comp.lang.python (http://bit.ly/comp-lang-python);
- comp.lang.python.announce (http://bit.ly/comp-lang-py-announce);
- r/python subreddit (http://www.reddit.com/r/python);
- Planet Python (http://planet.python.org/).

Наконец, рассмотрим сайты, с которых можно скачать разнообразные пакеты:

- The Python Package Index (https://pypi.python.org/pypi);
- Awesome Python (https://awesome-python.com/);

- Stack Overflow Python Questions (https://oreil.ly/S1vEL);
- 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 широкий диапазон типажей: энтузиастов, спорщиков, глупцов, хипстеров, интеллигентов — и множество других. Сообщество Python довольно дружелюбно. В зависимости от вашего местонахождения вы можете найти группы, увлекающиеся Python (http://python.meetup.com/). 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/) — 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).

Вакансии, связанные с Python

Чтобы найти вакансии, связанные с Python, воспользуйтесь следующими сайтами:

- Indeed (https://www.indeed.com/);
- Stack Overflow (https://stackoverflow.com/jobs);
- ZipRecruiter (https://www.ziprecruiter.com/candidate/suggested-jobs);
- Simply Hired (https://www.simplyhired.com/);
- CareerBuilder (https://www.careerbuilder.com/);
- Google (https://www.google.com/search?q=jobs);
- LinkedIn (https://www.linkedin.com/jobs/search).

На большей части этих сайтов введите слово `python` в первой строке поиска и ваше местоположение во второй. Среди хороших локальных сайтов могу порекомендовать Craigslist, эта ссылка, например, работает для Сиэтла (https://seattle.craigslist.org/search/jjj). Просто измените `seattle` на `sfbay`, `boston`, `nyc` или другой префикс сайта Craigslist, чтобы выполнить поиск в других районах. Для поиска вакансий Python, подразумевающих удаленную работу (на базе дистанционного доступа или «работу из дома»), воспользуйтесь следующими сайтами:

- Indeed (https://oreil.ly/pFQwb);
- Google (https://oreil.ly/LI529);
- LinkedIn (https://oreil.ly/nhV6s);

- Stack Overflow (https://oreil.ly/R23Tx);
- Remote Python (https://oreil.ly/bPW1I);
- We Work Remotely (https://oreil.ly/9c3sC);
- ZipRecruiter (https://oreil.ly/ohwAY);
- Glassdoor (https://oreil.ly/tK5f5);
- Remotely Awesome Jobs (https://oreil.ly/MkMeg);
- Working Nomads (https://oreil.ly/uHVE3);
- GitHub (https://oreil.ly/smmrZ).

Читайте далее

Но погодите, это еще не конец! В следующих трех главах вы можете узнать, как использовать 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 — бижутерия, а какие — серебряные пули. 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 таблиц для вещей, которые мне постоянно нужно подсматривать. Ваш мозг, скорее всего, лучше подготовлен, но так они всегда будут под рукой.

Упражнения

Питонщикам сегодня ничего не задавали.

ГЛАВА 20
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 чернослив, а не ревень.
>
> *Граучо Маркс*

В этой и двух следующих главах рассказывается о применении Python в наиболее распространенных областях деятельности человека: искусстве, бизнесе и науке. Если вас интересует одна из указанных областей, то можете почерпнуть из этих глав несколько идей или они подтолкнут вас попробовать что-то новое.

Двумерная графика

Все языки программирования в какой-то степени работают с компьютерной графикой. Многие мощные платформы в этом приложении в целях быстродействия были написаны на C или C++, но для продуктивности добавлены библиотеки Python. Начнем с рассмотрения некоторых библиотек для работы с двумерными изображениями.

Стандартная библиотека

В стандартной библиотеке содержатся всего несколько модулей, связанных с изображениями. 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`, то можете запустить этот код:

```
>>> import imghdr
>>> imghdr.what('oreilly.png')
'png'
```

Еще одна стандартная библиотека называется `turtle` (https://oreil.ly/b9vEz), она иногда используется для обучения молодых людей программированию. Вы можете запустить демонстрацию с помощью следующей команды:

```
$ python -m turtledemo
```

На рис. 20.1 показан снимок экрана, на котором демонстрируется пример *розетки*.

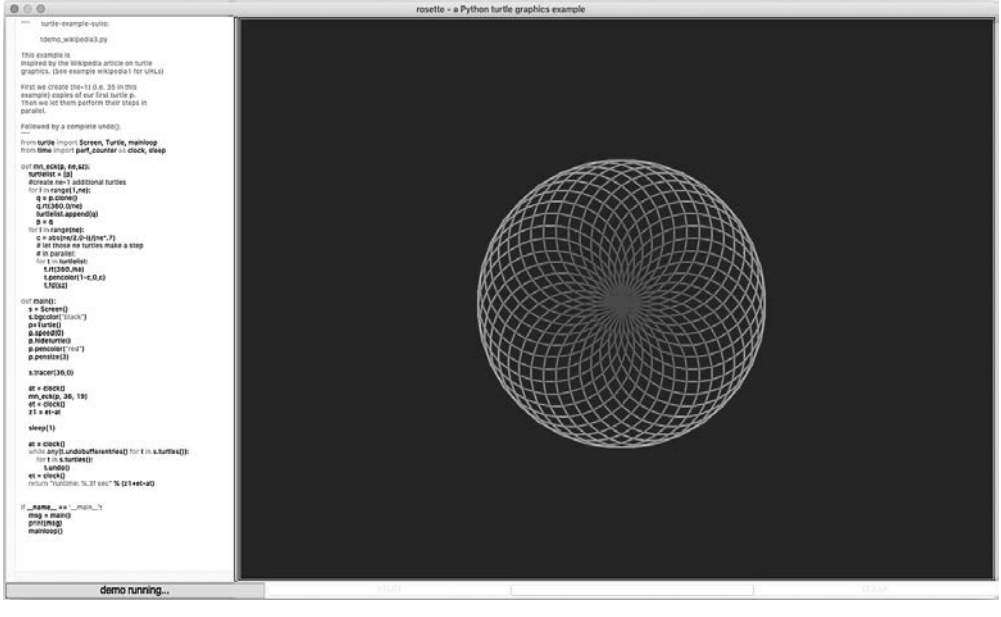

Рис. 20.1. Изображение из turtledemo

Чтобы сделать с графикой в Python нечто серьезное, нужно загрузить сторонние пакеты. Рассмотрим их.

PIL и Pillow

Многие годы библиотека изображений Python (Python Image Library, PIL) (http://bit.ly/py-image), несмотря на ее отсутствие в стандартной библиотеке, является самой известной библиотекой для обработки двумерных изображений. Она предшествовала таким установщикам, как `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`, то 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'
```

Хоть пакет и называется `Pillow`, вы импортируете его как `PIL`, чтобы код был совместим со старым `PIL`.

Чтобы отобразить изображение на экране с помощью метода `show()` объекта `Image`, вы сначала должны установить пакет ImageMagick, описанный в следующем подразделе, а затем попробовать вот что:

```
>>> img.show()
```

Изображение, показанное на рис. 20.2, 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.)

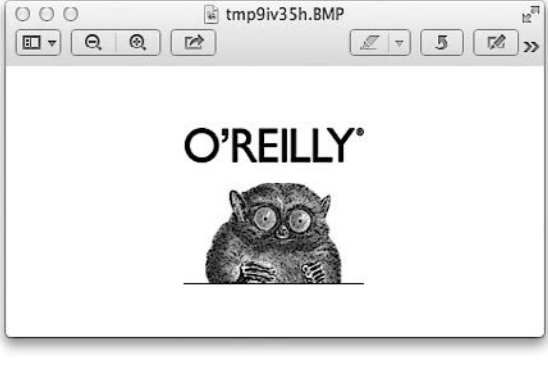

Рис. 20.2. Изображение, полученное с помощью библиотеки изображений Python

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* (origin), 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 — 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. 20.3.

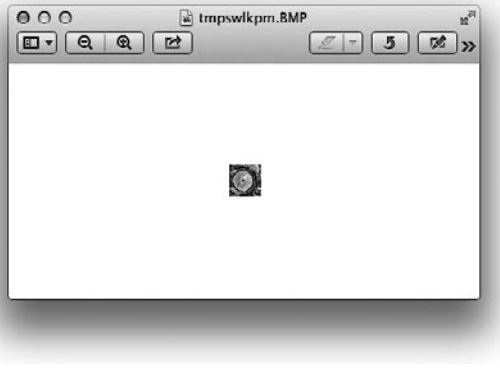

Рис. 20.3. Обрезанное изображение

Сохраним изображение с помощью метода `save()`. Он принимает имя файла и опциональный путь. Если у имени файла есть суффикс, то библиотека использует его, чтобы определить тип. Но вы также можете указать тип файла явно. Сохранить изображение с расширением GIF можно, проделав следующее:

```
>>> img2.save('cropped.gif', 'GIF')
>>> img3 = Image.open('cropped.gif')
>>> img3.format
'GIF'
>>> img3.size
(30, 30)
```

В качестве последнего примера «улучшим» наш маленький талисман. Сначала загрузим копию исходного изображения нашего зверька, показанную на рис. 20.4.

У него какая-то неопрятная тень, повернутая на пять часов, поэтому найдем изображение, которое позволит улучшить его образ (рис. 20.5).

Рис. 20.4. Наш милый зверек

Рис. 20.5. Инопланетная технология

Объединим их, используя магию *альфа*-канала, чтобы сделать пересечение полупрозрачным, как показано в примере 20.1.

Пример 20.1. ch20_critter.py

```
from PIL import Image

critter = Image.open('ch20_critter.png')
stache = Image.open('ch20_stache.png')
stache.putalpha(100)
img = Image.new('RGBA', critter.size, (255, 255, 255, 0))
img.paste(critter, (0, 0))
img.paste(stache, (45, 90), mask=stache)
img.show()
```

На рис. 20.6 показан новый образ зверька.

Рис. 20.6. Наш новый, более опрятный талисман

ImageMagick

ImageMagick (http://www.imagemagick.org/) — это комплект программ для конвертирования, изменения и отображения двумерных 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/). Чтобы установить ее, введите следующую команду:

```
$ pip install Wand
```

Библиотека `wand` позволяет делать примерно то же, что и 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)
```

Благодаря `wand` можно 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.

Трехмерная графика

Для работы с трехмерной графикой в Python существуют следующие пакеты:

- VPython (https://vpython.org/) содержит примеры (https://oreil.ly/J42t0), которые вы можете запустить в своем браузере;

- pi3d (https://pi3d.github.io/) работает на базе Raspberry Pi, Windows, Linux и Android;
- Open3D (http://www.open3d.org/docs) — это полноценная библиотека для работы с трехмерной графикой.

Трехмерная 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», чтобы увидеть действующие предложения.

Далее показаны некоторые пакеты Python для работы с 3D.

- Panda3D (http://www.panda3d.org/) — этот пакет имеет открытый исходный код, и его можно использовать бесплатно даже для создания коммерческих приложений. Вы можете скачать его версию с сайта Panda3D (http://bit.ly/dl-panda).
- VPython (https://vpython.org/) — поставляется с большим количеством примеров (https://oreil.ly/J42t0).
- Blender (http://www.blender.org/) — это бесплатное средство создания 3D-анимации и игр. Если вы скачаете и установите его (http://www.blender.org/download), то на ваш компьютер также будет установлена копия Python 3.
- Maya (https://oreil.ly/PhWn-) — это коммерческая система для создания 3D-анимации и графики. Она поставляется вместе с версией Python — в данный момент Python 2.7. Чед Вернон 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 скачать бесплатную версию, которая называется Apprentice. Как и другие пакеты для анимации, он поставляется с привязкой к Python (https://oreil.ly/L4C7r).

Графические пользовательские интерфейсы (GUI)

Название содержит слово «графический», но GUI (Graphical User Interface) 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, в которой 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. 20.7.

Рис. 20.7. Изображение, показанное с помощью библиотеки Tkinter

Чтобы окно пропало, нажмите кнопку Close (Закрыть) или выйдите из интерпретатора Python.

О библиотеке Tkinker вы можете прочитать в tkinter wiki (https://wiki.python.org/moin/TkInter). Теперь 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 норвежской компанией Trolltech. Использовался для создания таких приложений, как Google Earth, Maya и Skype. Кроме того, применен 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 (http://bit.ly/qt-diff). Скачать PySide можно 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). Скачать Qt бесплатно можно здесь: http://bit.ly/qt-dl.
- *GTK+* (http://www.gtk.org/). Соперник Qt, также был использован для создания множества приложений, таких как GIMP и оболочки Gnome для Linux. Для Python используется PyGTK (http://www.pygtk.org/). Чтобы скачать код, перейдите на сайт PyGTK (http://bit.ly/pygtk-dl), где 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 скачать 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). Включает поддержку мультитача. Вы можете скачать ее для всех платформ с сай-

та Kivy (http://kivy.org/#download). Библиотека содержит руководства по разработке приложений (http://bit.ly/kivy-intro).

- *PySimpleGUI* (https://pysimplegui.readthedocs.io/). Создайте нативный или основанный на веб интерфейс с помощью одной библиотеки. PySimpleGUI представляет собой оболочку для других интерфейсов, упомянутых в этом разделе, включая Tk, Kivy и Qt.
- *The Web*. Такие фреймворки, как Qt, используют встроенные компоненты, но некоторые другие программы применяют Web. Это универсальный GUI, содержащий графику (SVG), текст (HTML) и даже мультимедиа (в HTML5). 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, то клиент называется *толстым* или *насыщенным*; последний эпитет звучит более льстиво. Части приложения могут общаться друг с другом с помощью RESTful API, AJAX и JSON.

Диаграммы, графики и визуализация

Python — отличный инструмент для создания диаграмм, графиков и визуализации данных и особенно популярен в научной среде (см. главу 22). Полезные обзоры с примерами встречаются в официальной Python wiki (https://oreil.ly/Wdter) и Python Graph Gallery (https://python-graph-gallery.com/).

Рассмотрим самые популярные. В следующей главе вы снова увидите некоторые из них, но они будут использованы для создания карт.

Matplotlib

Бесплатную библиотеку для создания двумерных диаграмм Matplotlib (http://matplotlib.org/) можно установить с помощью следующей команды:

```
$ pip install matplotlib
```

Примеры из галереи (http://matplotlib.org/gallery.html) показывают широту этой библиотеки.

Попробуем написать такое же приложение для показа изображений (результаты можно увидеть на рис. 20.8) 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()
```

Реальная мощь Matplotlib заключается в построении *графиков* (plotting), что в целом понятно по средней части ее названия. Создадим два списка, каждый из

которых включает по 20 целых чисел. В один из них войдут числа, плавно увеличивающиеся от 1 до 20, а второй будет похож на первый, но с небольшими отклонениями там и тут (пример 20.2).

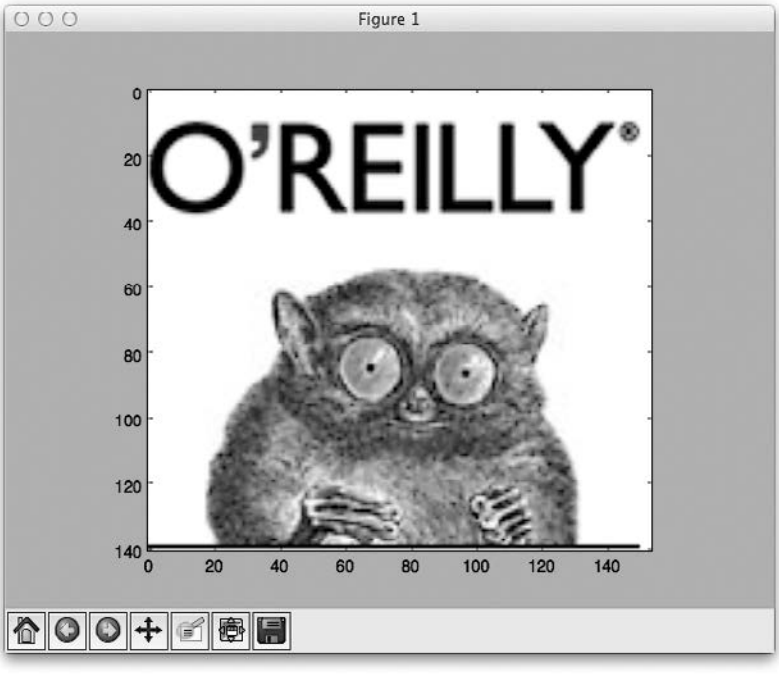

Рис. 20.8. Изображение, показанное с помощью библиотеки Matplotlib

Пример 20.2. ch20_matplotlib.py

```
import matplotlib.pyplot as plt
from random import randint

linear = list(range(1, 21))
wiggly = list(num + randint(-1, 1) for num in linear)

fig, plots = plt.subplots(nrows=1, ncols=3)

ticks = list(range(0, 21, 5))
for plot in plots:
    plot.set_xticks(ticks)
    plot.set_yticks(ticks)

plots[0].scatter(linear, wiggly)
plots[1].plot(linear, wiggly)
plots[2].plot(linear, wiggly, 'o-')

plt.show()
```

Если вы запустите эту программу, то получите результат, похожий на тот, что представлен на рис. 20.9 (он не будет полностью идентичен, поскольку каждый вызов функции `randint()` возвращает новый результат).

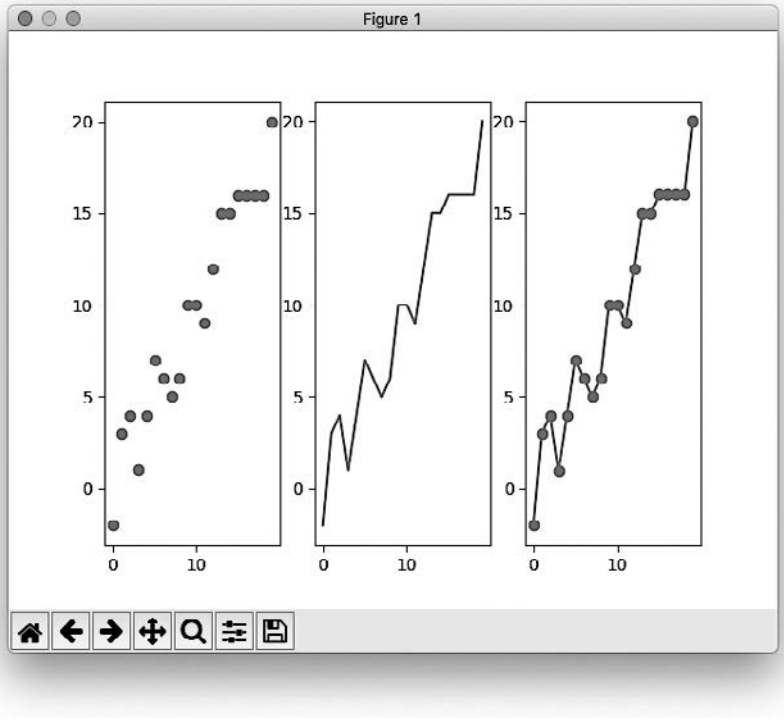

Рис. 20.9. Базовые графики Matplotlib (диаграмма рассеивания и линейные графики)

В этом примере показаны диаграмма рассеивания, линейный график и линейный график с отметками. При их создании были использованы стили по умолчанию, но внешний вид графиков можно изменять. Для получения более подробной информации обратитесь к сайту Matplotlib (https://matplotlib.org/) или к обзору, например *Python Plotting With Matplotlib (Guide)* (https://oreil.ly/T_xdT).

В главе 22 мы еще вернемся к Matplotlib — она тесно связана с NumPy и другими научными приложениями.

Seaborn

Seaborn (https://seaborn.pydata.org/) — это библиотека для визуализации данных (рис. 20.10), построенная на основе Matplotlib и имеющая соединение с Pandas. Для нее работает базовая мантра инсталляции (`pip install seaborn`).

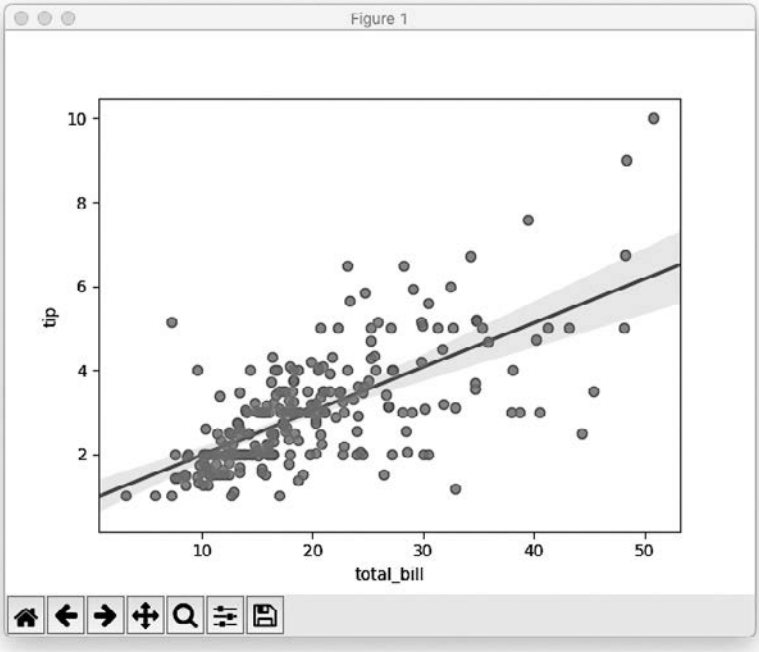

Рис. 20.10. Базовая диаграмма рассеивания и прямолинейная регрессия в Seaborn

Код, показанный в примере 20.3, основан на примере использования Seaborn (https://oreil.ly/eBFGi); он получает доступ к тестовым данным о чаевых в ресторане и строит график зависимости чаевых от общей суммы счета, добавляя прямолинейную регрессию.

Пример 20.3. ch20_seaborn.py

```
import seaborn as sns
import matplotlib.pyplot as plt

tips = sns.load_dataset("tips")
sns.regplot(x="total_bill", y="tip", data=tips);

plt.show()
```

При запуске данного кода с помощью стандартного интерпретатора Python вам понадобится указать первую строку импорта (import matplotlib.pyplot as plt) и последнюю строку (plt.show()), как показано в примере 20.3, иначе график не отобразится. Если вы используете Jupyter, то библиотека Matplotlib уже встроена в среду — нет необходимости ее импортировать. Помните это, когда будете исследовать примеры кода, работающего с инструментами построения карт в Python.

Как и Matplotlib, Seaborn предоставляет широкий спектр возможностей по обработке и отображению данных.

Bokeh

На заре существования Интернета разработчики генерировали графику на сервере и давали браузеру URL для доступа к ней. С недавних пор 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). Bokeh делает акцент на быстрой визуализации крупных наборов данных.

Если вы уже установили необходимые для Bokeh пакеты (NumPy, Pandas и Redis), то можете установить и сам этот инструмент, введя следующую команду:

```
$ pip install bokeh
```

(NumPy и Pandas в действии вы сможете увидеть в главе 22.)

Или же можете установить все сразу с сайта Bokeh (https://oreil.ly/1Fy-L). Несмотря на то что на сервере запущен Matplotlib, Bokeh в основном работает в браузере и может пользоваться всеми новыми возможностями клиентской стороны. Нажмите любое изображение в галерее (https://oreil.ly/DWN-d), чтобы получить интерактивное представление дисплея и его код Python.

Игры

Оказывается, Python — настолько хорошая платформа для написания игр, что об этом пишут книги, например:

- *Invent Your Own Computer Games with Python*, автор Эл Свайгарт (http://inventwithpython.com/)[1];
- *The Python Game Book*, автор Хорст Йенс (книга в формате docuwiki) (http://thepythongamebook.com/).

В Python wiki (https://wiki.python.org/moin/PythonGames) вы можете найти статью, в которой содержится еще большее количество ссылок.

Самая известная платформа для написания игр, скорее всего, pygame (http://pygame.org/). Вы можете скачать исполняемый установщик для своей платформы с сайта Pygame (http://pygame.org/download.shtml) и прочесть построчный пример создания игры (https://oreil.ly/l-swp).

---

[1] *Свайгарт Эл.* Учим Python, делая крутые игры. — М.: Эксмо, 2017.

Аудио и музыка

I sought the serif
But that did not suit Claude Debussy[1].

Deservedly Anonymous

Что насчет звука, музыки и котов, которые поют Jingle Bells? Ну, как говорит Meatloaf, два из трех — это не так плохо.

В напечатанной книге звук представить сложно, так что взгляните на эти актуальные ссылки на пакеты Python, предназначенные для работы со звуком и музыкой. С помощью Google вы можете найти намного больше:

- модули стандартной библиотеки для работы с аудио (http://docs.python.org/3/library/mm.html);
- сторонние инструменты для работы с аудио (https://wiki.python.org/moin/Audio);
- десятки сторонних музыкальных (https://wiki.python.org/moin/PythonInMusic) приложений: графические и CLI-проигрыватели, преобразователи, нотации, анализ, плейлисты, MIDI и др.

Наконец, как насчет онлайн-источников музыки? На протяжении этой книги вы видели примеры кода, которые получали доступ к Internet Archive. Вот ссылки на некоторые из аудио-архивов:

- аудиозаписи (https://archive.org/details/audio) (> 5 000 000);
- живая музыка (https://archive.org/details/etree) (> 200 000);
- живые концерты группы Grateful Dead (https://archive.org/details/GratefulDead) (> 13 000).

Читайте далее

Имитируем бурную деятельность! В дело вступает Python.

Упражнения

20.1. Установите `matplotlib`. Нарисуйте диаграмму рассеивания для следующих пар `(x, y)`: `( (0, 0), (3,5), (6, 2), (9, 8), (14, 10) )`.

20.2. Нарисуйте линейчатый график на основе тех же данных.

20.3. Нарисуйте график на основе тех же данных.

[1] Обыгрывается строка из песни I shot the Sheriff. — *Примеч. пер.*

ГЛАВА 21
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 прежде чем *взяться за дело*, он вешает пиджак на спинку стула, ослабляет галстук, закатывает рукава и наливает себе кофе. Тем временем женщина-бизнесмен уже выполняет свою работу. Может быть, возьмет латте.

В области бизнеса и правительства мы используем все технологии из предыдущих глав — базы данных, веб-системы и сети. Продуктивность Python делает его более популярным и для корпораций (http://bit.ly/py-enterprise), и для стартапов (http://bit.ly/py-startups).

Бизнес долго боролся с несовместимыми форматами файлов, скрытыми сетевыми протоколами, обязательными языками и общей нехваткой точной документации. Он может создавать более быстрые, более дешевые эластичные приложения, пользуясь следующими инструментами:

- динамическими языками, такими как Python;
- сетью как универсальным графическим интерфейсом пользователя;
- RESTful API как независимыми от языка интерфейсами сервисов;
- 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 Python не очень широко известны и в ряде случаев плохо задокументированы, они могут пригодиться. Рассмотрим некоторые из них, работающие 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 (http://bit.ly/py-excel) 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.

OpenOffice (http://openoffice.org/) — альтернатива Office, имеющая открытый исходный код. Работает в операционных системах Linux, Unix, Windows и macOS, а 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://oreil.ly/mLiCr) и библиотеки PyUNO (https://oreil.ly/FASNB).

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 просто. 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, то python-magic (https://pypi.org/project/python-magic/) может угадать его формат, проанализировав определенные последовательности байтов.

Библиотека python open document (http://appyframework.org/pod.html) позволяет предоставить код Python внутри шаблонов для создания динамических документов.

Формат PDF распространен также в области бизнеса, несмотря на то что 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, то 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-либо в строку поиска. Многие модули — это интерфейсы для общедоступных API различных сервисов. 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, то это результат действий покупателя или полтергейст? Cubes (http://cubes.databrewery.org/) — это веб-сервер и браузер данных Online Analytical Processing (OLAP);
- OpenERP (https://www.openerp.com/) — крупная коммерческая система Enterprise Resource Planning (ERP), 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 весьма своеобразно, словно для того, 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 требуются 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 могут взаимодействовать 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, то требуется иметь достаточное количество памяти. Традиционное программное обеспечение для этого не подходит. Таким компаниям, как Google и Amazon, пришлось изобрести решения, позволяющие работать с данными такого масштаба. Netflix (http://bit.ly/py-netflix) — 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 в первую очередь. Если говорить языком бизнеса, распространенными терминами являются «извлечение», «преобразование», «загрузка» (extract, transform, load, или ETL). Синонимы наподобие *«выпас данных»* (data munging, data wrangling) могут создать впечатление, будто вы приручаете непокорного зверя, — такая метафора

может оказаться близкой. Данная задача выглядит решенной с точки зрения инженерии, но по большей части это искусство. Я немного говорил об этом в главе 12. Мы рассмотрим *науку о данных* более подробно в главе 22, поскольку именно на эту область разработчики тратят бóльшую 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 будучи моложе, я думал: «И она говорит об этом только сейчас?!» Хотя теперь 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, например словари и объекты, тысячи стандартных и сторонних библиотек, а также сообщество экспертов, которое можно найти в поисковике.

Если вы программист, пишущий программы для бизнеса, то ваш рабочий процесс практически всегда содержит следующее.

1. Извлечение данных из файлов странных форматов или баз данных.
2. Очистку данных, которая охватывает множество областей, заполненных подводными камнями.
3. Преобразование дат, времени и наборов символов.
4. Выполнение каких-либо действий над данными.
5. Сохранение полученного результата в базе данных.
6. Откат к шагу 1: 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 16. Или же найти модуль, который считывает непосредственно бинарный формат электронной таблицы. Ваши пальцы знают, как набрать строку `python excel` в поисковике и найти сайты наподобие 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 решить ее снова, но уже используя библиотеки, которые позволяют сэкономить немного времени. Мы считаем CSV-файл, агрегируем все числа в одном столбце, упорядочив их по уникальному значению, и выведем результат на экран. Если бы мы решали задачу с помощью SQL, то применили 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 врач). Мы используем модуль `csv`, который рассматривали в подразделе «CSV» на с. 328, и счетчик `Counter` из подраздела «Подсчитываем элементы с помощью Counter()» на с. 233. 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. Элемент `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 инструментария 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, то 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, вы можете заметить: пример с `bubbles` использовал один вызов функции (`aggregate`) с целью заменить чтение и подсчет данных в формате CSV вручную. В зависимости от того, что вам нужно, наборы инструментов для работы с данными могут сберечь 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. Найти хорошие образцы практических задач, связанных с данными, и их решений на Python я бы порекомендовал в следующих источниках:

- *Data Crunching: Solve Everyday Problems Using Java, Python, and More* Грега Уилсона (http://bit.ly/data_crunching) (издательство Pragmatic Bookshelf);
- *Automate the Boring Stuff* Эла Свайгарта (https://automatetheboringstuff.com/)[1] (издательство No Starch).

Инструменты очистки данных могут сэкономить много времени, и Python имеет множество таких инструментов. Например, PETL (http://petl.readthedocs.org/) позволяет выполнять извлечение и переименование рядов и колонок. В главе 22 рассматриваются особенно полезные инструменты для работы с данными: Pandas,

[1] *Свайгарт Э.* Автоматизация рутинных задач с помощью Python. Практическое руководство для начинающих. — Киев: Диалектика, 2018.

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.

Вы также можете вернуться назад и взглянуть на инструменты для сериализации и валидации данных, рассмотренные в разделе «Сериализация данных» на с. 386.

Валидация данных

При очистке данных вам зачастую нужно проверить следующее:

- тип данных (целое число, число с плавающей точкой или строка);
- диапазон возможных значений;
- корректность значений, например рабочий номер телефона или адрес электронной почты;
- дубликаты;
- отсутствующие данные.

Это особенно важно при обработке веб-запросов и ответов.

Для работы с определенными данными можно использовать следующие полезные пакеты Python:

- `validate_email` (https://pypi.org/project/validate_email);
- `phonenumber` (https://pypi.org/project/phonenumbers).

Кроме того, порекомендую полезные инструменты общего назначения:

- `validators` (https://validators.readthedocs.io/);
- `pydantic` (https://pydantic-docs.helpmanual.io/) — для версий Python 3.6 и выше; использует подсказки типов;
- `marshmallow` (https://marshmallow.readthedocs.io/en/3.0) — также выполняет сериализацию и десериализацию;
- `Cerberus` (http://docs.python-cerberus.org/en/stable);
- многие другие (https://libraries.io/search?keywords=validation&languages=Python).

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 (https://www.data.gov/developers/apis) созданы на основе CKAN (http://ckan.org/), 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* (https://froide.readthedocs.io/) — 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.

Пакеты для работы с бизнес-данными с открытым исходным кодом

- *Odoo* (https://www.odoo.com/) — обширная ERP-платформа.
- *Tryton* (http://www.tryton.org/) — еще одна обширная платформа.
- *Oscar* (http://oscarcommerce.com/) — фреймфорк для электронной коммерции, написанный на Django.
- *Grid Studio* (https://gridstudio.io/) — основанные на Python таблицы, работают локально или в облаке.

Python в области финансов

С недавнего времени в финансовой индустрии развился значительный интерес к Python. Адаптируя программное обеспечение, показанное в главе 22, а также разрабатывая собственное, группа *quants* создает новое поколение финансовых инструментов:

- *Quantitative economics* (http://qeconomics.org/ojs/index.php/) — этот инструмент предназначен для экономического моделирования с помощью большого количества математических расчетов и Python;
- *Python for finance* (http://www.python-for-finance.com/) — этот сайт представляет книгу *Derivatives Analytics with Python: Data Analytics, Models, Simulation, Calibration and Hedging* Ива Хилпиша (издательство Wiley);
- *Quantopian* (https://www.quantopian.com/) — это интерактивный сайт, на котором вы можете писать собственный код Python и работать с его помощью с архивными данными 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* (http://bit.ly/python-finance)[1] (издательство O'Reilly) — книга Ива Хилпиша, содержащая примеры финансового моделирования, написанные на Python.

Безопасность бизнес-данных

Безопасность — особая забота для бизнеса. Данной теме посвящены целые книги, поэтому я лишь упомяну несколько советов, связанных с Python.

- В подразделе «scapy» раздела «TCP/IP» главы 17 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 таблицы.
- Книга *Violent Python* (http://bit.ly/violent-python) (с подзаголовком *A Cookbook for Hackers, Forensic Analysts, Penetration Testers and Security Engineers*) Ти Джея О'Коннора (издательство Syngress) — это широкий обзор Python и компьютерной безопасности.

Карты

Карты стали полезными для многих бизнесов. Python очень хорошо рисует карты, так что мы потратим не много времени на эту тему. Менеджеры любят графики, и если вы можете быстро нарисовать красивую карту сайта вашей организации, то это вам не повредит.

В ранние дни существования Интернета я посещал экспериментальный сайт по созданию карт у Xerox. Когда появились крупные сайты, такие как Google Maps, они стали откровением (к тому же вызывали мысль: «Почему я не подумал об этом и не заработал миллионы?»). Теперь сервисы картографии и сервисы, *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 многое другое. 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 (помогающие этой горилле). Многие из этих сервисов имеют интерфейсы Python. Поговорим о некоторых из них, начиная с самых простых форматов.

---

[1] *Хилпиш И.* Python для финансовых расчетов. — Киев: Диалектика, 2019.

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 данного формата содержит несколько файлов, включающих как минимум следующую информацию:

- `.shp` — информацию о фигуре (вектор);
- `.shx` — индекс формы;
- `.dbf` — базу данных атрибутов.

Получим шейп-файл для нашего следующего примера. Посетите страницу http://bit.ly/cultural-vectors. В разделе Admin 1 — States, Provinces (Admin 1 — штаты, провинции) нажмите зеленую кнопку download states and provinces (Скачать штаты и провинции), чтобы скачать архив. Затем 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 на основе шейп-файла

В этом подразделе приводится очень упрощенная демонстрация чтения и отображения шейп-файла. Вы увидите, что результат неидеален, и поймете преимущества работы с пакетами более высокого уровня, которые будут показаны ниже.

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
    # единицы измерения карты
    mwidth = mright - mleft
    mheight = mtop - mbottom
```

```
    # преобразуем единицы измерения карт в единицы измерения изображений
    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, соединяющий 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. 21.1.

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 19 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 использовать 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).

Насколько я знаю, не существует пакета для работы с картами, основанного на чистом Python. К счастью, есть более изящные пакеты; рассмотрим их.

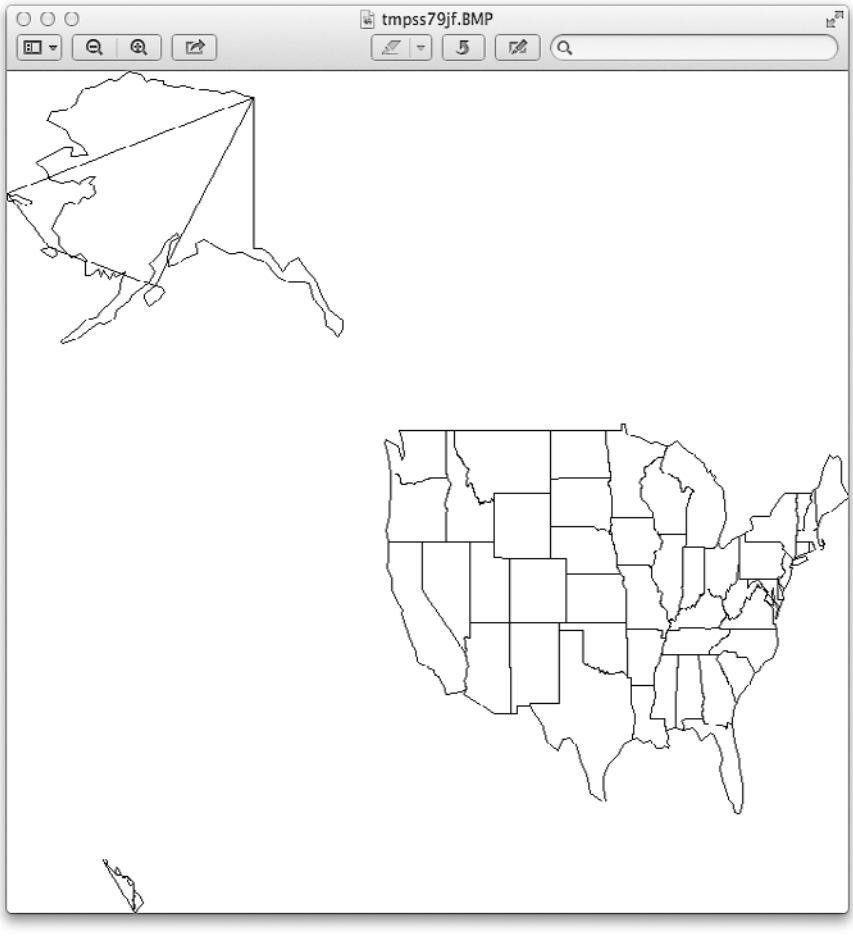

Рис. 21.1. Предварительная карта

Geopandas

Пакет Geopandas (http://geopandas.org/) интегрирует в себе Matplotlib, Pandas и другие библиотеки Python в одну платформу, которая позволяет работать с геопространственными данными.

Основной пакет устанавливается с помощью знакомой команды `pip install geopandas`, но полагается и на другие пакеты, которые вам также нужно устанавливать, используя `pip`, если их у вас еще нет:

- `numpy`;
- `pandas` (версия 0.23.4 или выше);
- `shapely` (интерфейс для GEOS);

- `fiona` (интерфейс для GDAL);
- `pyproj` (интерфейс для PROJ);
- `six`.

Пакет Geopandas может читать шейп-файлы (включая файлы из предыдущего подраздела), а также, что удобно, содержит два шейп-файла от Natural Earth: очертания стран/континентов и столицы. В примере 21.1 используются оба этих файла.

Пример 21.1. geopandas.py

```
import geopandas
import matplotlib.pyplot as plt

world_file = geopandas.datasets.get_path('naturalearth_lowres')
world = geopandas.read_file(world_file)
cities_file = geopandas.datasets.get_path('naturalearth_cities')
cities = geopandas.read_file(cities_file)
base = world.plot(color='orchid')
cities.plot(ax=base, color='black', markersize=2)
plt.show()
```

Запустив этот пример, вы получите карту, показанную на рис. 21.2.

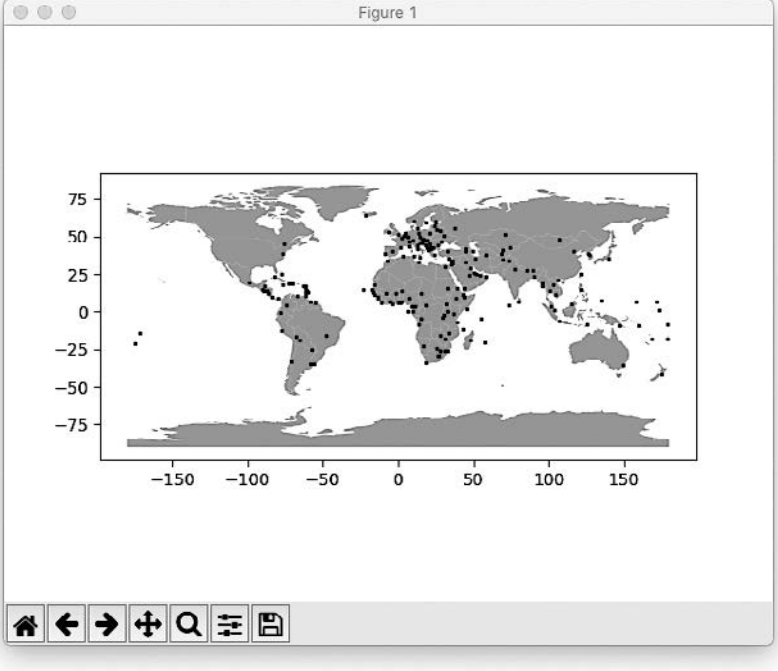

Рис. 21.2. Карта, созданная с помощью Geopandas

Я считаю пакет Geopandas наилучшей комбинацией для управления географическими данными и их отображения. Однако существует множество других стоящих конкурентов, которых мы рассмотрим в следующем подразделе.

Другие пакеты для работы с картами

Перед вами набор ссылок на другое ПО для работы с картами; многие из них могут не полностью устанавливаться с помощью `pip`, но для некоторых можно задействовать `conda` (альтернативный установщик пакетов Python, особенно полезный для научного ПО):

- *pyshp* (https://pypi.org/project/pyshp) — это библиотека для работы с шейп-файлами, упомянутая выше, в подразделе «Нарисуем карту на основе шейп-файла»;
- *kartograph* (http://kartograph.org/) отрисовывает шейп-файлы в карты формата SVG на сервере или клиенте;
- *shapely* (https://shapely.readthedocs.io/) решает геометрические вопросы наподобие «Какие строения в этом городе через 50 лет окажутся в зоне наводнения?»;
- *basemap* (http://matplotlib.org/basemap) основана на Matplotlib, предназначена для рисования карт и перекрытия их данных; к сожалению, после появления Cartopy считается устаревшей;
- *cartopy* (https://scitools.org.uk/cartopy/docs/latest) продолжает дело Basemap и решает задачи, для которых обычно нужен пакет Geopandas;
- *folium* (https://python-visualization.github.io/folium) работает с `leaflet.js`, используется `geopandas`;
- *plotly* (https://plot.ly/python/maps) — еще один пакет для построения, который включает в себя возможность работы с графиками;
- *dash* (https://dash.plot.ly/) использует Plotly, Flask и JavaScript для создания интерактивных визуализаций, включая карты;
- *fiona* (https://github.com/Toblerity/Fiona) оборачивает библиотеку OGR, которая работает с шейп-файлами и другими векторными форматами;
- *Open Street Map* (https://oreil.ly/BJeha) получает доступ к обширной коллекции карт мира OpenStreetMap (https://www.openstreetmap.org/);
- *mapnik* (http://mapnik.org/) — библиотека, написанная на C++, имеющая привязку к Python; используется для создания векторных (линии) и растровых (изображения) карт;
- *Vincent* (http://vincent.readthedocs.org/) преобразуется в Vega, инструмент визуализации JavaScript; см. руководство *Mapping data in Python with pandas and vincent* (https://oreil.ly/0TbTC);
- *Python for ArcGIS* (http://bit.ly/py-arcgis) — ссылки на ресурсы Python для коммерческого продукта ArcGIS фирмы Esri;
- *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* (http://bit.ly/py-geo-dev)[1] (издательство Packt) — книга Эрика Вестры, содержащая примеры использования `mapnik` и других инструментов;
- *Learning Geospatial Analysis with Python* (http://bit.ly/learn-geo-py) (издательство Packt) — еще одна книга. Ее написал Джоэл Лохед. Он сделал обзор форматов и библиотек, а также включил геопространственные алгоритмы;
- *geomancer* (https://github.com/thinkingmachines/geomancer) полезен для геопространственных работ, например для нахождения расстояния от некой точки до ближайшего ирландского бара.

Если вам интересны карты, то попробуйте скачать и установить один из этих пакетов и посмотреть, что вы можете сделать с его помощью. Или можете избежать установки ПО и попытаться соединиться с API удаленного сервера самостоятельно — в главе 18 показывается, как 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 (https://oreil.ly/Zqw0W) (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);
- googlemaps (http://py-googlemaps.sourceforge.net/).

Если вы авторизуетесь с помощью Google или другого источника, чтобы получить ключ для API, то сможете получить доступ к другим сервисам, 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.

---

[1] *Вестра Э.* Разработка геоприложений на языке Python. — М.: ДМК Пресс, 2017.

Нам следует упомянуть здесь Data Science Toolkit (http://www.datasciencetoolkit.org/). Он бесплатно предоставляет возможности двунаправленного геокодирования, вычисления координат политических границ и статистики и многое другое. Вы можете загрузить все данные и ПО как виртуальную машину и запустить их отдельно на своем компьютере.

Читайте далее

Мы отправимся на научную ярмарку и посмотрим все экспонаты, связанные с Python.

Упражнения

21.1. Установите Geopandas и запустите пример 21.1. Попробуйте поизменять цвета и размеры меток.

ГЛАВА 22
Python в науке

Во времена ее владения сила пара
Стала величайшей на суше и воде,
И теперь все полагаются
На последние достижения науки[1].

Джеймс Макинтайр. Ода к юбилею королевы, 1887

В последние годы в основном из-за ПО, показанного в этой главе, Python стал очень популярен среди ученых. Если вы и сами ученый или студент, то, возможно, пользовались такими инструментами, как MatLab и R, или традиционными языками, например Java, C или C++. Сейчас 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 ряд особенностей и модулей, которые мы проигнорировали.

Математические функции

Python имеет множество математических функций в стандартной библиотеке math (https://oreil.ly/01SHP). 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
```

[1] Оригинальная цитата: In her reign the power of steam On land and sea became supreme, And all now have strong reliance In fresh victories of science. — *Примеч. пер.*

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 на основании `e` с помощью функции `log()`:

```
>>> math.log(1.0)
0.0
>>> math.log(math.e)
1.0
```

Если вы хотите задать другое основание логарифма, то 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 этот оператор 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 повторить ее быстро три раза, не начав плеваться), то библиотека `math` вдобавок предоставит вам 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, то 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` (https://oreil.ly/1EZQ0).

Рассчитываем точное значение чисел с плавающей точкой с помощью модуля decimal

Числа с плавающей точкой в вычислительной технике не похожи на настоящие числа, которые мы изучали в школе:

```
>>> x = 10.0 / 3.0
>>> x
3.3333333333333335
```

Эй, что это за пятерка в конце? На ее месте должна быть тройка. Так происходит потому, что количество битов в регистрах процессора ограничено и числа, которые не являются степенями двойки, не могут быть выражены точно.

С помощью модуля `decimal` (https://oreil.ly/o-bmR) вы можете записывать числа с желаемым уровнем точности. Это особенно важно для расчетов денежных сумм. Сумма в валюте Соединенных Штатов не может быть меньше одного цента (сотая часть доллара), поэтому при подсчете количества долларов и центов нам нужно быть предельно точными. Если мы попробуем представить доллары и центы как значения с плавающей точкой, такие как 19,99 или 0,06, то потеряем некоторую часть точности в последних битах еще до начала вычислений. Как с этим справиться? Легко. 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 его ближайшее целое значение:

```
>>> penny = Decimal('0.01')
>>> total.quantize(penny)
Decimal('21.19')
```

Такие же результаты доступны 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 возможен ущерб.

Выполняем вычисления для рациональных чисел с помощью модуля fractions

Вы можете представлять числа как числитель, разделенный на знаменатель, с помощью модуля стандартной библиотеки Python `fractions` (https://oreil.ly/l286g). 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(33333333333333333333333333333, 100000000000000000000000000000)
```

Получим наибольший общий делитель для двух чисел с помощью функции `gcd`:

```
>>> import fractions
>>> fractions.gcd(24, 16)
8
```

Используем Packed Sequences с помощью модуля array

В Python список больше похож на связанный список, а не на массив. Если вы хотите получить одномерную последовательность элементов одинакового типа, то применяйте тип `array` (https://oreil.ly/VejPU). 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 (такой как `int` или `float`), а опциональный *`инициализатор`* содержит исходные значения, которые можно передать как список, строку или итерируемое значение.

Я никогда не использовал этот пакет для решения реальных задач. Это низкоуровневая структура данных, полезная для представления чего-то наподобие изображений. Если для выполнения числовых подсчетов вам на самом деле нужен массив, особенно имеющий больше одного измерения, то лучше задействовать NumPy, о котором мы поговорим чуть позже.

Обрабатываем простую статистику с помощью модуля statistics

Начиная с версии Python 3.4, `statistics` (https://oreil.ly/DELnM) — стандартный модуль. Он содержит традиционные функции: среднее, медиану, режим, 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, доступных в пакетах, таких как SciPy и Pandas, которые мы рассмотрим далее в этой главе.

Перемножение матриц

Начиная с Python 3.5, вы увидите символ `@`, который делает необычные вещи. Он все еще может использоваться для декораторов, но будет применяться также и для *перемножения матриц* (https://oreil.ly/fakoD). Если вы работаете со старой версией Python, то вам следует задействовать NumPy.

Python для науки

В оставшейся части этой главы рассматриваются сторонние пакеты Python для науки и математики. Несмотря на то что вы можете установить их индивидуально, следует рассмотреть их одновременное скачивание в качестве части научного дистрибутива Python. Рассмотрим основные варианты.

- *Anaconda* (https://www.anaconda.com/) — это бесплатный пакет, имеющий множество самых свежих возможностей. Он поддерживает Python 2 и 3 и не вредит установленной у вас версии Python.
- *Enthought Canopy* (https://assets.enthought.com/downloads) — доступна как бесплатная, так и коммерческая версии.
- *Python(x, y)* (https://python-xy.github.io/) — этот релиз подходит только для Windows.
- *Pyzo* (http://www.pyzo.org/) — этот пакет основан на некоторых инструментах пакета Anaconda, а также других.

Я рекомендую установить пакет Anaconda. Он большой, и все представленное в этой главе содержится в нем. Примеры остальной части главы подразумевают, что вы установили необходимые пакеты либо отдельно, либо как часть Anaconda.

NumPy

NumPy (http://www.numpy.org/) — это одна из основных причин популярности Python среди ученых (http://www.numpy.org/). Вы слышали, что динамические языки, такие как Python, зачастую медленнее компилирующих языков наподобие C или даже

других интерпретируемых языков, например Java. NumPy был написан для предоставления доступа к быстрым многомерным массивам по аналогии с научными языками, такими как FORTRAN. Вы получаете скорость C и дружелюбную к разработчикам природу Python.

Если вы скачали один из научных дистрибутивов Python, то у вас уже есть NumPy. В противном случае следуйте инструкциям на странице скачивания NumPy (https://oreil.ly/HcZZi).

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, двумерный — на таблицу со строками и столбцами, а трехмерный — на кубик Рубика. Длина измерений может не быть одинаковой.

Массивы array в NumPy и array в Python — не одно и то же. Далее в этой главе 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 помощью 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.

Создаем массив 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,)
```

Создаем массив с помощью функции arange()

Метод `arange()`NumPy похож на стандартный метод `range()` Python. Если вы вызовете метод `arange()`, передав ему один целочисленный аргумент `num`, то 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` диктует функции `arrange()`, какой тип значения следует создать:

```
>>> g = np.arange(10, 4, -1.5, dtype=np.float)
>>> g
array([ 10. ,   8.5,   7. ,   5.5])
```

Создаем массив с помощью функций zeros(), ones() и random()

Метод `zeros()` возвращает массив, все значения которого равны `0`. В эту функцию вам нужно передать аргумент, содержащий желаемую форму 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. Одно из их различий — 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, то используйте индексы, разделенные запятыми, которые заключены в квадратные скобки:

```
>>> a.shape = (2, 5)
>>> a
array([[0, 1, 2, 3, 4],
       [5, 6, 7, 8, 9]])
>>> a[1,2]
7
```

Это отличается от двумерного списка, в котором индексы лежат в разных парах квадратных скобок:

```
>>> 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` столбцам (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 нечто более полезное. Для начала мы задействуем 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, то 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 и 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 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. Чтобы напечатать меньше текста, вы также можете дать NumPy указание найти *скалярное произведение* массивов:

```
>>> product = np.dot(coefficients, answers)
>>> product
array([ 20.,  13.])
```

Если решение верно, то значения массива `product` должны быть близки к значениям массива `dependents`. Функция `allclose()` позволяет 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 математических и статистических функций. Релиз SciPy (https://oreil.ly/Yv7G-) 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 области, с которыми работает также коммерческий MATLAB (https://oreil.ly/jOPMO) или приложение с открытым исходным кодом R (http://www.r-project.org/).

Библиотека SciKit

Как и предыдущая библиотека, SciKit (https://scikits.appspot.com/) — это группа научных пакетов, построенная на основе SciPy. SciKit-Learn (https://scikit-learn.org/) — это пакет, который специализируется на *машинном обучении*: поддерживает моделирование, классификацию, кластеризацию и разнообразные алгоритмы.

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 науку о данных, инструменты, о которых я говорил ранее, — NumPy, SciPy и собственно Pandas, вынесенный в тему этого подраздела, — компоненты растущего популярного инструментария, работающего с данными. (Mac и Сан-Франциско опциональны.)

Pandas (http://pandas.pydata.org/) — это новый пакет для интерактивного анализа данных. Он особенно полезен для манипулирования данными реального мира с помощью комбинирования матричной математики NumPy и возможности обработки таблиц и реляционных баз данных. В книге *Python for Data Analysis: Data Wrangling with Pandas, NumPy and IPython* Уэса Маккини[1] (издательство O'Reilly) рассматривается 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, называемую `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 можно встретить не только в науке, но и в бизнесе.

[1] *Маккини У.* Python и анализ данных. — М.: ДМК-Пресс, 2015.

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 скачать 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. Python помогает им быстрее стать более продуктивными.

Python и научные области

Мы рассматривали инструменты Python, которые можно использовать 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 несколько узконаправленных библиотек.

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/comp-phys-py);
- Astropy (https://www.astropy.org/);
- SunPy (https://sunpy.org/) (анализ данных, поступающих с Солнца);
- MetPy (https://unidata.github.io/MetPy) (анализ метеорологических данных);
- Py-ART (https://arm-doe.github.io/pyart) (погодный радар);
- Community Intercomparison Suite (http://www.cistools.net/) (атмосферные науки);
- Freud (https://freud.readthedocs.io/) (анализ траектории);
- Platon (https://platon.readthedocs.io/) (атмосферы экзопланет);
- PSI4 (http://psicode.org/) (квантовая химия);
- OpenQuake Engine (https://github.com/gem/oq-engine);
- yt (https://yt-project.org/) (волюметрический анализ данных).

Биология и медицина:

- Biopython (https://biopython.org/);
- Python для биологов (http://pythonforbiologists.com/);
- Введение в прикладную биоинформатику (http://readiab.org/);

- Нейровизуализация с помощью Python (http://nipy.org/);
- MNE (https://www.martinos.org/mne) (визуализация нейрофизиологических данных);
- PyMedPhys (https://pymedphys.com/);
- Nengo (https://www.nengo.ai/) (симулятор нейронов).

Проводятся следующие международные конференции по Python и научным данным:

- PyData (http://pydata.org/);
- SciPy (http://conference.scipy.org/);
- EuroSciPy (https://www.euroscipy.org/).

Читайте далее

Мы достигли конца наблюдаемой вселенной Python, за исключением приложений из мультивселенной.

Упражнения

22.1. Установите Pandas. Получите CSV-файл в примере 16.1. Запустите программу из примера 16.2. Поэкспериментируйте с командами Pandas.

Приложения

ПРИЛОЖЕНИЕ А

Аппаратное и программное обеспечение для начинающих программистов

Некоторые вещи понятны на интуитивном уровне. Что-то мы видим в природе, что-то изобрел человек — например, колесо или пиццу.

В другие вещи нужно просто поверить. Как телевидение преобразует невидимые колебания в воздухе в звук и движущиеся изображения?

Компьютер попадает во вторую категорию. Как добиться того, чтобы машина поняла и выполнила напечатанное вами?

Когда я учился программированию, было трудно найти ответы на какие-то простые вопросы. Например, в некоторых книгах память компьютера объясняется через аналогию с библиотечной книжной полкой. Мне казалось: если вы *читаете данные из памяти*, то в рамках аналогии берете книгу с полки. Означает ли это, что книга удаляется из памяти? На самом деле нет. Это больше похоже на получение *копии* книги с полки.

В данном приложении приводится краткий обзор аппаратного и программного обеспечения компьютера, если вы относительно недавно начали программировать. Я попробую объяснить то, что в будущем станет очевидным, но сейчас эта информация может стать хорошей отправной точкой.

Аппаратное обеспечение

Компьютеры пещерных людей

Когда пещерные люди Ог и Тог возвращались с охоты, они подходили к своим кучкам камней и клали в каждую из них по одному камню за каждого убитого мамонта. Но эти кучи годились, только чтобы похвастаться тем, что у одного из них камней больше, чем у другого.

Далекие потомки Ога (Тог был растоптан мамонтом, когда клал очередной камень) научились считать, писать и пользоваться абаком. Но, чтобы перейти от этих инструментов к концепции компьютера, потребуется задействовать воображение и технологические достижения. Первая необходимая технология — электричество.

Электричество

Бен Франклин считал, что электричество — это поток некой невидимой жидкости из места с большим количеством жидкости *(положительного)* в место с меньшим количеством *(отрицательное)*. Он был прав, но перепутал термины. Электроны путешествуют от «отрицательного» места в «положительное», но они были открыты гораздо позже — менять терминологию оказалось слишком поздно. Поэтому с тех пор нам нужно помнить, что электроны движутся в одну сторону, а *ток* течет в другую.

Все мы знакомы с такими природными электрическими феноменами, как статическое электричество или молнии. После того как люди открыли способ отправить электроны по проводам для создания электрических *цепей*, мы еще на один шаг приблизились к созданию компьютеров.

Раньше я думал, что электрический ток в проводе вызывается проворными электронами, которые нарезают круги по всей дистанции. На самом деле это не так. Электроны перемещаются от одного атома к другому. Они ведут себя как шарикоподшипники в трубке (или шарики тапиоки в соломинке). Добавленный с одного конца шарик толкает соседа. Так происходит до тех пор, пока шарик с противоположного конца не вытолкнется. Как правило, электроны перемещаются медленно (*скорость перемещения* по проводу составляет примерно три дюйма в час), однако это практически мгновенное столкновение очень быстро распространяет электромагнитную волну: ее скорость составляет от 50 до 99 % скорости света в зависимости от проводника.

Изобретения

Нам все еще нужно следующее:

- способ запоминать какие-то данные;
- способ что-то сделать с теми данными, которые мы запомнили.

Одной из концепций памяти был *выключатель* — предмет, который либо включен, либо выключен, при этом он сохраняет свое состояние до тех пор, пока что-то не переключит его в другое состояние. Электрический выключатель работает, открывая и закрывая цепь, позволяя электронам течь по ней или блокируя их. Мы постоянно используем выключатели, чтобы управлять светом и другими электрическими устройствами. Нужен был способ управлять самим выключателем с помощью электричества.

Самые первые компьютеры (и телевизоры) использовали для этих целей вакуумные трубки, но те были довольно большими и часто перегорали. Ключевым изобретением, приведшим к появлению современных компьютеров, явился *транзистор*: маленький, эффективный и надежный. Финальным аккордом стало значительное уменьшение размеров транзисторов и объединение их в *интегральные схемы*.

С течением лет компьютеры становились быстрее и дешевели по мере того, как их размеры уменьшались. Сигналы перемещаются быстрее, когда компоненты находятся рядом друг с другом.

Однако существует предел тому, насколько маленьким можно сделать транзистор. Резвость электрона встречает *сопротивление*, вследствие чего вырабатывается тепло. Мы достигли предела более десяти лет назад, производители компенсируют это, размещая несколько чипов на одной плате. Как следствие, повысился спрос на *распределенные вычисления*; мы рассмотрим данное явление позже.

Независимо от этих деталей, с помощью рассмотренных изобретений мы смогли собрать *компьютеры*: машины, которые могут запоминать данные и что-то делать с ними.

Идеальный компьютер

Реальные компьютеры имеют множество сложных особенностей. Сконцентрируемся на самых необходимых компонентах.

Интегральная плата содержит центральный процессор, память и провода, соединяющие их друг с другом и другими устройствами.

Процессор

ЦП (центральный процессор), или чип, выполняет собственно вычисления:

- математические задачи вроде сложения;
- сравнение значений.

Память и кэш

ОЗУ (оперативное запоминающее устройство) занимается «запоминанием». Оно работает быстро, но *энергозависимо* (данные теряются, если выключается питание).

ЦП становятся еще быстрее, чем память, поэтому разработчики компьютеров добавляют *кэши*: небольшие и более быстрые участки памяти, располагающиеся между процессором и основной памятью. Пытаясь считать байты из памяти, ваш ЦП сначала обращается к ближайшему кэшу (он называется *L1*), затем к следующему (*L2*) и, наконец, к основной памяти.

Хранение

Поскольку основная память теряет данные, нам также нужно *энергонезависимое* хранилище. Такие устройства дешевле, чем память, и хранят гораздо больше данных, но и работают медленнее.

Традиционный метод хранения — *магнитные диски* (или *жесткие диски — hard drives*, *HDD*) с перемещающимися головками, которые отвечают за чтение и запись, как виниловые пластинки и стилус.

Гибридная технология называется *SSD* (*Solid State Drive — твердотельный накопитель*). Он изготавливается из полупроводников, как и ОЗУ, но при этом энергонезависим, как магнитные диски. Цена и скорость работы выше, чем у HDD, но ниже, чем у ОЗУ.

Ввод данных

Как вы помещаете данные в компьютер? Для людей основные варианты — это клавиатуры, мыши и сенсорные экраны.

Вывод данных

Как правило, люди видят данные с помощью экранов и принтеров.

Относительное время доступа

Время, которое требуется для записи и получения данных из этих компонентов, может значительно различаться. Это оказывает большое практическое влияние. Например, ПО должно работать в памяти и получать оттуда данные, но также должно надежно сохранять их в энергонезависимых устройствах, таких как диски. Проблема заключается в том, что диски в тысячи раз медленнее, а сеть — и того больше. Это значит, что программисты тратят много времени на выбор наилучшего компромиссного решения между скоростью и стоимостью.

В статье Computer Latency at a Human Scale (https://oreil.ly/G36qD) Дэвид Джеппесен сравнивает устройства. Я перенес его изыскания и другие исследования в табл. А.1. Последние столбцы — соотношение, относительное время (ЦП — одна секунда) и относительное расстояние (ЦП — один дюйм) — помогут лучше понять эту таблицу.

Таблица А.1. Относительное время доступа

Расположение	Время	Соотношение	Относительное время	Относительное расстояние
ЦП	0,4 нс	1	1 с	1 дюйм
Кэш L1	0,9 нс	2	2 с	2 дюйма
Кэш L2	2,8 нс	7	7 с	7 дюймов
Кэш L3	28 нс	70	1 мин	6 футов
ОЗУ	100 нс	250	4 мин	20 футов
SSD	100 мкс	250 000	3 дня	4 мили
Магнитный диск	10 мс	25 000 000	9 месяцев	400 миль
Интернет: SF→NY	65 мс	162 500 000	5 лет	2500 миль

Хорошо, что выполнение инструкции процессора на самом деле занимает меньше наносекунды, поскольку в противном случае за то время, которое потребовалось бы для обращения к диску несколько раз, у вас мог бы родиться ребенок. Время отклика диска и сети гораздо больше, чем для ЦП и ОЗУ, поэтому полезно выполнять в памяти максимально много работы. И поскольку процессор сам по себе быстрее, чем ОЗУ, лучше иметь последовательные данные, чтобы байты можно было отправить в более быстрые (и менее объемные) кэши, расположенные ближе к ЦП.

Программное обеспечение

Как же управлять всем этим оборудованием? Для начала нужны *инструкции* (команды, которые указывают процессору, что делать) и *данные* (входные и выходные параметры инструкций). В *компьютерах с запоминаемой программой* данными можно считать все, что упрощает процесс проектирования. Но как представить инструкции и данные? Что, если они сохраняются в одном месте, а выполняются в другом? Дальние родственники пещерного человека Ога очень хотели бы узнать это.

Вначале был бит

Вернемся к идее *выключателя* — чего-то, что сопровождает одно из двух значений. Он может быть либо выключен, либо включен, иметь либо высокое, либо низкое напряжение, быть положительным или отрицательным — он получает значение, не забывает его и может предоставить любому, кто запросит это значение. Интегральные схемы дали нам способ интегрировать и соединить миллиарды маленьких переключателей в маленькие чипы.

Если переключатель может иметь всего два значения, то его можно использовать для того, чтобы представить *бит*, или бинарную цифру. Ее значения можно расценивать как `0` и `1`, да и нет, `true` и `false` или любым другим удобным для нас способом.

Однако биты слишком малы, чтобы представить какое-либо значение, помимо `0` и `1`. Как нам убедить биты представлять более крупные объекты?

Чтобы получить ответ, взгляните на свои пальцы. Мы используем всего десять цифр (от 0 до 9) в повседневной жизни, а числа больше 9 получаются благодаря *позиционной записи*. Если я добавлю 1 к числу 38, то 8 станет 9 и мы получим значение 39. Если я добавлю еще 1, то 9 превратится в 0, а 1 *перенесется влево*, увеличивая 3 до 4, что даст нам результат, равный 40. Крайнее правое число — это «колонка единиц», то, что слева от него, — «колонка десятков» и т. д., по мере продвижения влево выполняется умножение на 10. С помощью трех десятичных чисел вы можете представить тысячу (10 * 10 * 10) чисел от `000` до `999`.

Мы можем использовать позиционную запись и для битов, объединяя их в более крупные коллекции. *Байт* имеет 8 бит и может представлять 2^8 (256) возможных комбинаций. Вы можете использовать байт для сохранения, например, небольших чисел от `0` до `255` (в позиционной записи нужно оставить место для нуля).

Байт выглядит как последовательность из 8 бит, каждый из которых имеет либо значение `0` (или «выкл.», или `false`), либо значение `1` (или «вкл.», или `true`). Самый правый бит — *наименее значимый*, а самый левый — *наиболее значимый*.

Машинный язык

ЦП каждого компьютера разработан с набором инструкций, представленных масками битов (они еще называются *кодами операций*), которые он может понимать. Каждый код операции выполняет определенную функцию; ее входные значения находятся в одном месте, а выходные — в другом. В процессоре есть специальные внутренние локации, называемые *регистрами*, в которых хранятся коды операций и значения.

Воспользуемся упрощенным компьютером, который работает только с байтами и имеет четыре регистра размером 1 байт, которые называются `A`, `B`, `C` и `D`. Предположим следующее:

- код операции команды помещается в регистр `A`;
- команда получает входные значения из регистров `B` и `C`;
- команда сохраняет результат в регистре `D`.

(Сложение 2 байт может вызвать *переполнение* регистра, но в рамках данного примера мы это проигнорируем.)

Предположим, что:

- в регистре `A` содержится код операции *сложения двух целых чисел*: десятичная `1` (бинарное представление — `00000001`);
- в регистре `B` содержится десятичное значение `5` (бинарное представление — `00000101`);
- в регистре `C` содержится десятичное значение `3` (бинарное представление — `00000011`).

ЦП видит, что в регистре `A` появилась инструкция. Он декодирует и запускает ее, читая значения из регистров `B` и `C` и передавая их во внутренние цепи, которые могут выполнять сложение. При получении результата мы должны увидеть десятичное значение `8` (бинарное представление — `00001000`) в регистре `D`.

Процессор выполняет сложение и другие математические функции, используя регистры похожим способом. Он *декодирует* код операции и перенаправляет управление конкретным цепям внутри ЦП. Он также может сравнивать объекты, например, «Больше ли значение регистра `B`, чем значение регистра `C`?» Кроме того, немаловажно, что он *получает* значения из памяти в процессоре и *сохраняет* значения из него в памяти.

Компьютер сохраняет *программы* (инструкции на машинном языке и данные) в памяти и обрабатывает передачу инструкций и данных в ЦП и из него.

Ассемблер

На машинном языке программировать сложно. Вам нужно идеально указывать каждый бит, что занимает очень много времени. Поэтому люди придумали чуть более читаемый уровень языков, которые называются *ассемблерными языками* или просто *ассемблерами*. Для каждого варианта процессора существует свой ассемблер. Такие языки позволяют использовать имена переменных, чтобы определить поток инструкций и данные.

Высокоуровневые языки

Чтобы освоить ассемблер, требуется приложить немало усилий, поэтому люди разработали более *высокоуровневые языки*, которые им было бы проще использовать. Такие языки транслируются в ассемблер программой, которая называется *компилятором*, или же запускаются непосредственно с помощью *интерпретатора*.

Одними из самых старых подобных языков являются FORTRAN, LISP и C — их дизайн и предполагаемое использование значительно различаются, но объединяет эти языки их место в компьютерной архитектуре.

Для реальных задач, как правило, выделяют определенные «стеки» ПО:

- *Мейнфрейм* — IBM, COBOL, FORTRAN и др.;
- *Microsoft* — Windows, ASP, C#, SQL Server;
- *JVM* — Java, Scala, Groovy;
- *системы с открытым исходным кодом* — Linux, языки (Python, PHP, Perl, C, C++, Go), базы данных (MySQL, PostgreSQL), веб (apache, nginx).

Программисты обычно остаются в одном из этих миров, используя языки и инструменты внутри него. Некоторые технологии, такие как TCP/IP и веб, позволяют этим стекам взаимодействовать.

Операционные системы

Каждая инновация построена на других новшествах, появившихся до нее, и обычно мы не знаем (или нам неважно), как работают более низкие уровни. Инструменты строят инструменты для создания еще большего количества инструментов, и мы принимаем это как должное.

Рассмотрим основные операционные системы:

- *Windows (Microsoft)* — коммерческая, много версий;
- *macOS (Apple)* — коммерческая;
- *Linux* — открытый исходный код;
- *Unix* — много коммерческих версий, в основном замененных на Linux.

Операционная система состоит из следующих компонентов:

- *ядро* — планирует выполнение программ и управляет ими, а также вводом/выводом;
- *драйверы устройств* — используются ядром для получения доступа к ОЗУ, диску и другим устройствам;
- *библиотеки* — файлы исходного кода и бинарные файлы, используемые разработчиками;
- *приложения* — отдельные программы.

Один компьютер может поддерживать больше одной операционной системы, но за один раз запускается только одна. Процесс запуска ОС называется *начальной загрузкой* (booting[1]), перезапуск — *перезагрузкой* (rebooting). Эти термины появились даже в области маркетинга фильмов — студии «перезагружают» предыдущие

[1] Это отсылает к выражению «тянуть себя за шнурки ботинок» (lifting yourself by your own bootstraps), что выглядит так же невероятно, как и компьютер.

неудачные попытки. Вы можете выполнить *мультисистемную загрузку* вашего компьютера, установив больше одной операционной системы, но запускать одновременно можно только одну.

Выражение *«пустая машина»* означает один компьютер, на котором запущена операционная система. В нескольких следующих подразделах мы отступим от «пустой машины».

Виртуальные машины

Операционная система — по сути, большая программа, поэтому в конечном счете кто-то придумал способ запустить чужие операционные системы как *виртуальные машины* (гостевые программы) на своих *хост*-машинах. Вы можете установить на свой компьютер Microsoft Windows, а затем в то же время запустить Linux на виртуальной машине, не покупая второй компьютер или не выполняя мультисистемную загрузку.

Контейнеры

Более свежая идея — это *контейнер*. Контейнер — способ запускать множество операционных систем одновременно до тех пор, пока у них одинаковое ядро. Идея была популяризована командой разработчиков Docker (https://www.docker.com/) — они взяли малоизвестные особенности ядра Linux и добавили к ним полезные возможности по управлению. Их аналогия по отправке контейнеров (которая революционизировала отправку и сэкономила нам много денег) была прозрачной и привлекательной. Выложив код в открытый доступ, команда разработчиков Docker добилась того, что контейнеры быстро приняла вся компьютерная отрасль.

Google и другие облачные провайдеры добавляли поддержку ядра Linux годами и использовали контейнеры в своих дата-центрах. Контейнеры используют меньше ресурсов, чем виртуальные машины, что позволяет упаковывать больше программ в одну коробку физического компьютера.

Распределенные вычисления и сети

Когда бизнес только начал применять ПК, нужно было найти способ заставить компьютеры общаться друг с другом, а также с другими устройствами, например принтерами. Изначально использовалось проприетарное сетевое ПО, такое как Novell, но в конечном счете его заменил стек протоколов TCP/IP, поскольку Интернет заметно развился во второй половине 1990-х. Microsoft позаимствовала стек TCP/IP из бесплатного варианта Unix, которая называется BSD[1].

Одним из эффектов интернет-бума стал спрос на *серверы*: машины и ПО, которые позволяют запустить все сервисы для работы с вебом, чатами и электронной почтой. Старый стиль *сисадмина* (системного администрирования) состоял в том,

[1] В некоторых файлах Microsoft вы можете найти записи об авторских правах, принадлежащих Университету Калифорнии.

чтобы устанавливать все программное обеспечение и оборудование и управлять им вручную. Спустя некоторое время стало понятно, что требуется автоматизация. В 2006 году Билл Бейкер из компании Microsoft придумал аналогию *«домашние животные против крупного рогатого скота»* (pets versus cattle) для управления серверами, и с тех пор она стала мемом в нашей отрасли (иногда ее формулируют как pets versus livestock) (табл. А.2).

Таблица А.2. Домашние животные против скота

Домашние животные	**Скот**
Именуются индивидуально	Нумеруются автоматически
Требуется индивидуальный подход	Стандартизированы
Восстанавливаются при сбоях	Заменяются при сбоях

Вы можете встретить наследника «сисадмина» — термин DevOps (development and operations — «разработка и использование»). Он связан с набором приемов, призванных поддержать быстрые изменения в сервисах, не вызвав в них перегрузок. Облачные сервисы могут быть очень крупными и сложными, и даже у крупных компаний, таких как Amazon или Google, время от времени случаются сбои.

Облако

Люди много лет создавали компьютерные *кластеры*, используя разные технологии. Одной из более ранних концепций был *Beowulf-кластер*: идентичные компьютеры для массовой обработки данных (от фирмы Dell или аналогов вместо рабочих станций от Sun или HP), связанные локальной сетью.

Термин *облачные вычисления* означает использование компьютеров в дата-центрах для выполнения вычислений и хранения данных — однако не только для тех компаний, которым принадлежат эти ресурсы бэкенда. Сервисы предоставляются всем, а стоимость зависит от времени работы ЦП, занятого объема дисков и т. д. Компания Amazon и ее *AWS* (Amazon Web Services) — самый известный облачный сервис, но *Azure* (Microsoft) и *Google Cloud* также довольно велики.

На самом деле эти облака представляют собой пустые машины, виртуальные машины и контейнеры — все они считаются скотом, а не домашними животными.

Kubernetes

Все компании, которым нужно управлять крупными кластерами компьютеров, расположенных в нескольких дата-центрах, — например, Google, Amazon и Facebook — создали или позаимствовали решения, которые позволяют этим компаниям масштабироваться.

- *Развертывание*. Как вы вводите в строй новое оборудование и программное обеспечение? Как вы заменяете его в случае сбоя?
- *Конфигурация*. Как должны работать эти системы? Им нужны такие вещи, как имена и адреса других компьютеров, пароли и настройки безопасности.

- *Оркестрация*. Как управлять всеми этими компьютерами, виртуальными машинами и контейнерами? Можно ли выполнить масштабирование, чтобы соответствовать уменьшению или увеличению нагрузки?
- *Обнаружение сервисов*. Как узнать, кто что делает и где находится?

Некоторые конкурирующие решения были созданы командой разработчиков Docker и другими. Но за последние несколько лет стало ясно, что эту битву выигрывает Kubernetes (http://kubernetes.io/).

Компания Google разработала крупные внутренние фреймворки для менеджмента под кодовыми именами Borg и Omega. Когда сотрудники компании предложили открыть исходный код этих «бриллиантов», руководство на некоторое время задумалось, но затем все же решилось. Google выпустила Kubernetes версии 1.0 в 2015 году, и с тех пор его экосистема и влияние постоянно растут.

ПРИЛОЖЕНИЕ Б
Установка Python 3

Большинство примеров этой книги были написаны и протестированы для Python 3.7, последней стабильной версии на момент ее написания. 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 большое количество способов установить новую версию. В данном приложении я опишу несколько из них.

- Стандартный установщик загружает Python с сайта python.org и добавляет вспомогательные программы, такие как `pip` и `virtualenv`.
- Если ваша работа связана с наукой, то вам лучше подойдет версия Python, объединенная с научными пакетами из Anaconda, — в таком случае вам нужно будет использовать установщик `conda` вместо `pip`.

Windows не содержит Python, а в macOS, Linux и Unix, как правило, содержатся старые версии. Пока они не наверстают упущенное, вам нужно будет установить Python 3 самостоятельно.

Проверьте свою версию Python

В консоли или окне терминала введите команду `python -V`:

```
$ python -V
Python 3.7.2
```

В зависимости от вашей операционной системы, если у вас нет Python или ОС не может его найти, вы увидите сообщение об ошибке, например command not found.

Если у вас есть Python версии 2.x, то вы можете захотеть установить Python 3 — либо для всей системы, либо только для себя в `virtualenv` (см. подраздел «virtualenv» на с. 434 или раздел «Установка virtualenv» данного приложения ниже). В этом приложении я покажу, как установить Python 3 для всей системы.

Установка стандартной версии Python

Перейдите в браузере на официальную страницу скачивания Python (https://www.python.org/downloads). Она попробует определить вашу операционную систему и предоставить подходящие вам варианты. Если она ошибется, то вы можете использовать следующие ссылки:

- версии Python для Windows (https://www.python.org/downloads/windows);
- версии Python для macOS (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. Б.1.

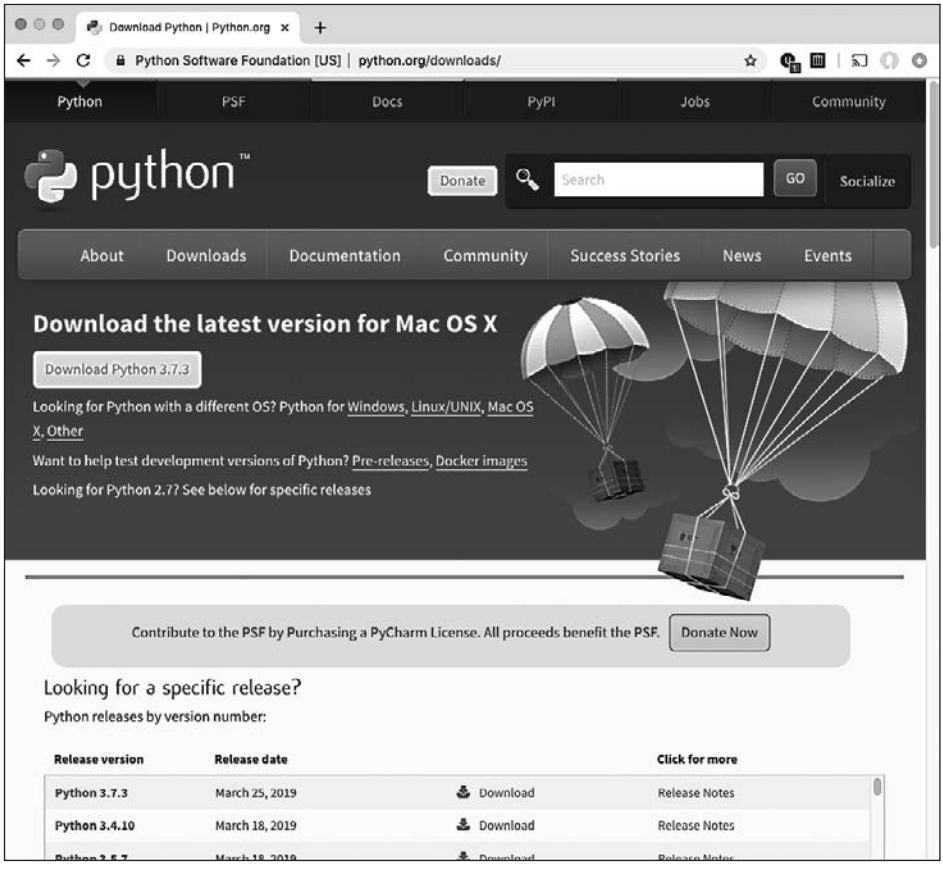

Рис. Б.1. Пример страницы загрузки

Нажав желтую кнопку Download Python 3.7.3 (Скачать Python 3.7.3), вы скачаете эту версию для своей ОС. Если вы хотите сначала немного узнать об этой версии,

то нажмите синюю ссылку Python 3.7.3 в первом столбце таблицы снизу, под строкой Release version. Так вы попадете на страницу информации, похожую на ту, что показана на рис. Б.2.

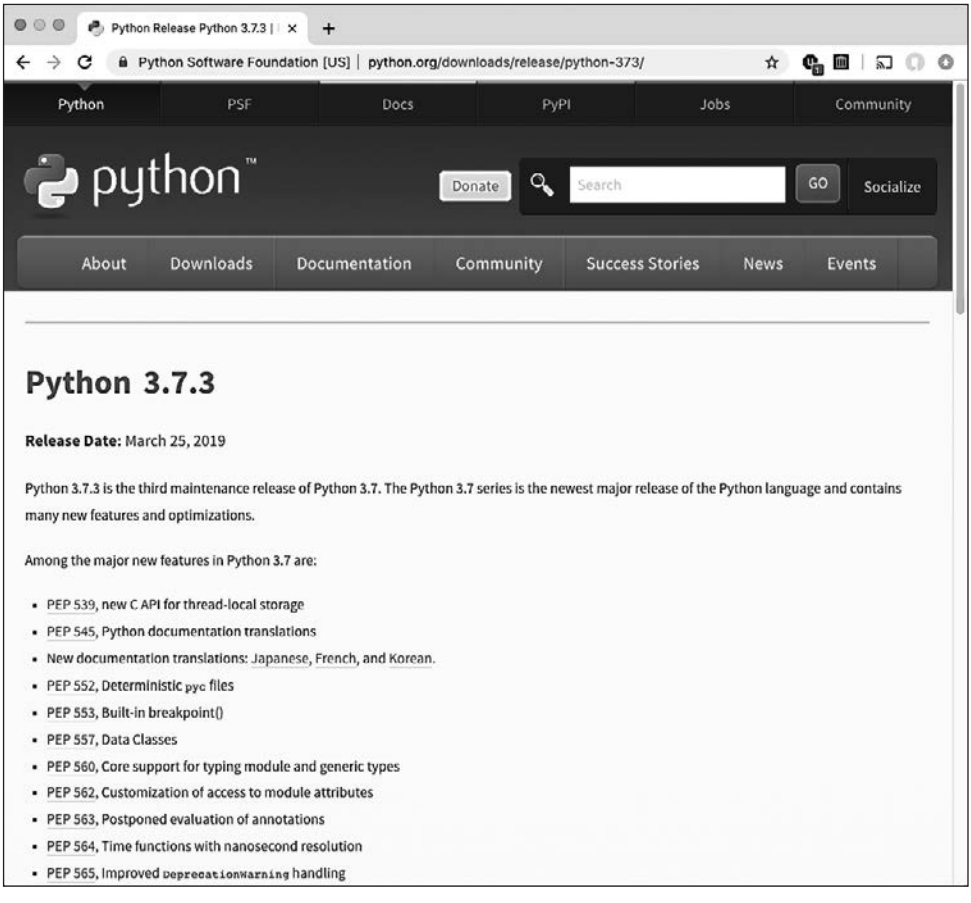

Рис. Б.2. Страница подробностей загрузки

Вам нужно прокрутить страницу вниз, чтобы увидеть ссылки для скачивания (рис. Б.3).

macOS

Нажмите ссылку macOS 64-bit/32-bit installer (https://oreil.ly/54lG8), чтобы скачать файл с расширением `.pkg` для Mac. Дважды щелкните на нем, чтобы увидеть начальное диалоговое окно (рис. Б.4).

Нажмите кнопку Continue (Продолжить). Вы увидите последовательность других диалоговых окон.

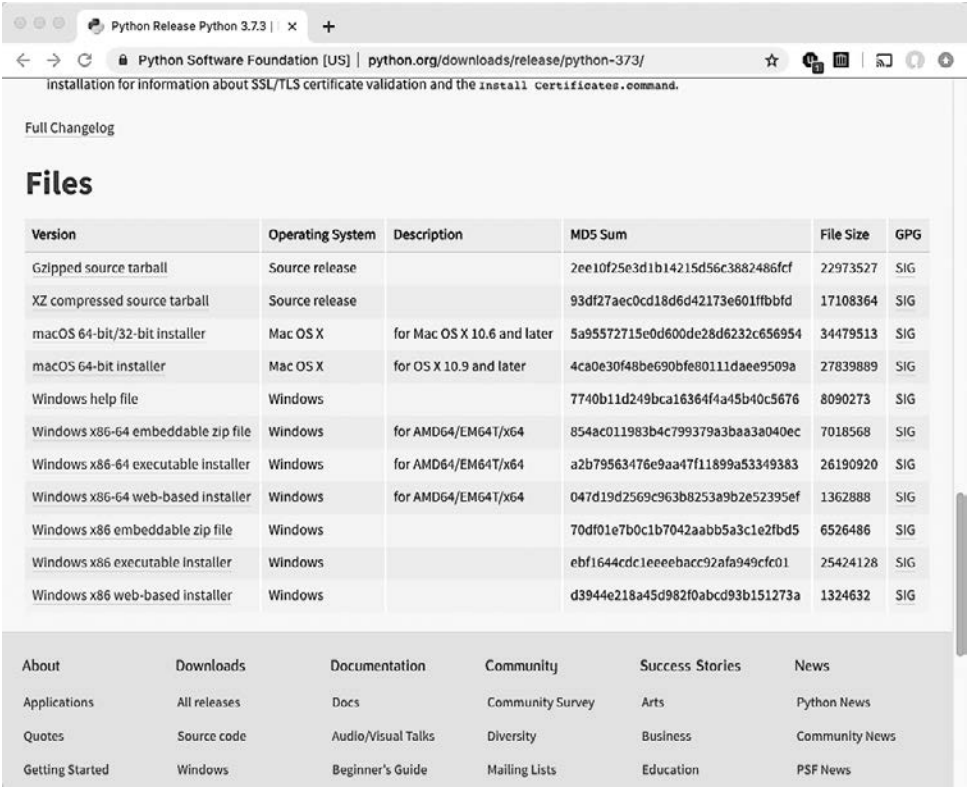

Рис. Б.3. Нижняя часть страницы, предлагающая скачать Python

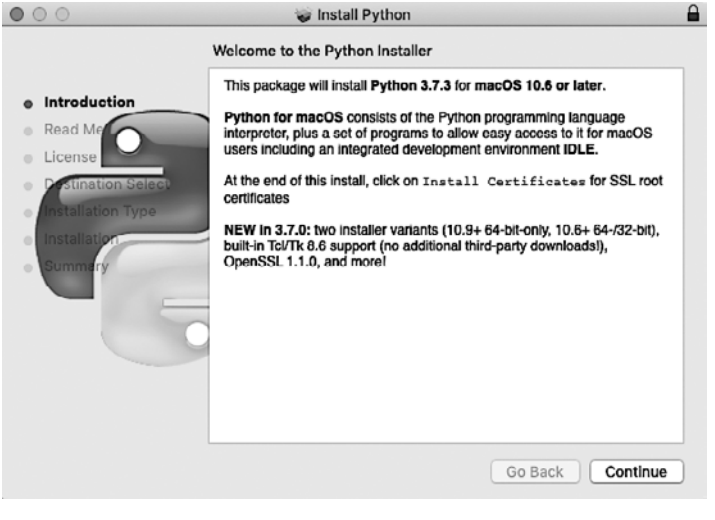

Рис. Б.4. Первый экран установочного диалога для Mac

Когда все будет готово, вы увидите диалог, показанный на рис. Б.5.

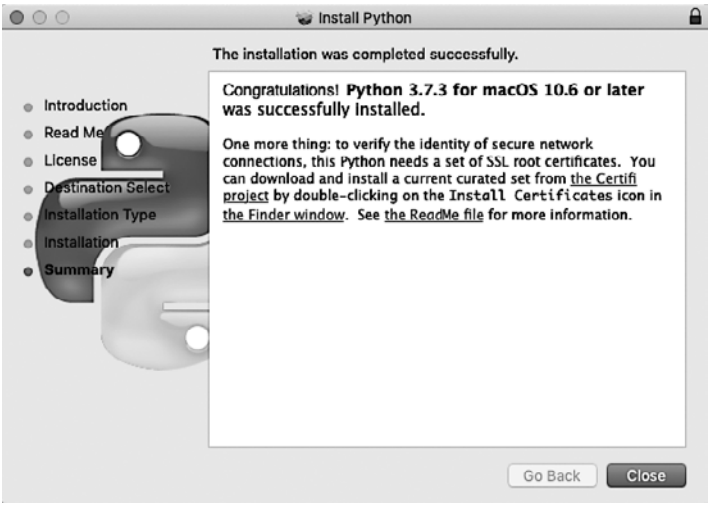

Рис. Б.5. Девятый экран установочного диалога для Mac

Python 3 будет установлен в каталог /usr/local/bin/python3, оставляя нетронутой существующую версию Python 2 на вашем компьютере.

Windows

ОС Windows никогда не включала в себя Python, но с недавних пор упростила его установку. Обновление Windows 10 от мая 2019 года (https://oreil.ly/G8Abf) включает в себя файлы `python.exe` и `python3.exe`. Это не интерпретаторы Python, а ссылки на новую страницу в Microsoft Store, посвященную Python 3.7 (https://oreil.ly/Lky_h). Вы можете использовать эту ссылку, чтобы скачать и установить Python так же, как вы это делаете для другого ПО.

Или же вы можете скачать и установить Python с официального сайта:

- 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 установлена у вас (32- или 64-битная), сделайте следующее:

- нажмите кнопку Пуск;
- щелкните правой кнопкой мыши на пункте Мой компьютер;
- выберите пункт меню Свойства и найдите битовое значение.

Щелкните на соответствующем установщике (файл с расширением `.msi`). После того как он будет скачан, нажмите его дважды 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).

Скачайте 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 менеджера пакетов pip

Если вы вышли за пределы стандартной установки Python, то два инструмента практически бесценны для разработки на Python: это `pip` и `virtualenv`.

Пакет `pip` — это самый популярный способ установить сторонние (нестандартные) пакеты Python. Несколько раздражало то, что такой полезный инструмент не являлся частью стандартного Python и его приходилось скачивать и устанавливать самостоятельно. Как говорил мой друг, это жестокий, пугающий ритуал. Хорошая новость заключается в том, что, начиная с версии 3.4, `pip` — стандартная часть Python.

Если у вас установлен Python 3, но под рукой только версия `pip` для Python 2, то получить версию для Python 3 под Linux или macOS можно так:

```
$ 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.

Установка virtualenv

Программа `virtualenv` зачастую применяется вместе с `pip` и представляет собой способ установить пакеты Python в определенный каталог (папку), чтобы избежать пересечений с пакетами Python, уже существующими в системе. Это позволяет использовать любые пакеты Python, даже если у вас нет разрешения изменять существующие установленные пакеты.

Ниже представлена пара хороших руководств по `pip` и `virtualenv`:

- A Non-Magical Introduction to Pip and Virtualenv for Python Beginners (http://bit.ly/jm-pip-vlenv);
- The Hitchhiker's Guide to Packaging: Pip (http://bit.ly/hhgp-pip).

Другие способы работы с пакетами

Как вы уже видели, техники упаковки пакетов в Python различаются, и ни одна из них не может решить сразу все задачи. PyPA (https://www.pypa.io/) (Python Packaging Authority) — это группа добровольцев (не являющаяся частью официальной группы по разработке ядра Python), которая работает над тем, чтобы упростить взаимодействие с пакетами Python. Группа написала руководство Python Packaging User's Guide (https://packaging.python.org/), где рассматриваются известные проблемы и их решения.

Самые популярные инструменты — `pip` и `virtualenv`, я использовал их на протяжении всей этой книги. Если они не могут вам помочь или же вы хотите попробовать что-то новое, то вот несколько альтернатив:

- `pipenv` (https://pipenv.readthedocs.io/) объединяет в себе `pip` и `virtualenv`, а также включает и другие возможности. Обратите внимание на критику, которой он подвергался (https://oreil.ly/NQ3pH), и обсуждение (https://oreil.ly/psWa-);
- `poetry` (https://poetry.eustace.io/) — это конкурент, который решает некоторые проблемы `pipenv`.

Но самой главной альтернативой, особенно если речь идет о научных или работающих с большими данными приложениях, является `conda`. Вы можете получить ее как часть дистрибутива Anaconda, о котором я поговорю далее, или же отдельно (см. раздел «Устанавливаем Anaconda» ниже).

Устанавливаем Anaconda

Anaconda (https://docs.anaconda.com/anaconda) — это всеобъемлющий установщик с акцентом на науку. Последняя версия, Anaconda3, включает в себя Python 3.7 и его стандартную библиотеку, а также язык R для приложений, посвященных науке о данных. Среди других плюсов — библиотеки, о которых мы говорили ранее в этой книге: beautifulsoup4, Flask, ipython, Matplotlib, nose, numpy, Pandas, pillow, pip, scipy, tables, zmq и множество других. Anaconda содержит кросс-платформенную программу установки, которая называется `conda`, мы рассмотрим ее в следующем пункте.

Чтобы установить Anaconda 3, посетите страницу скачивания (https://www.anaconda.com/distribution) версий Python 3. Нажмите соответствующую ссылку для вашей платформы (номера версий изменились с момента написания данного текста, но вы сможете в этом разобраться):

- установщик для macOS установит все в каталог `anaconda`, расположенный под вашим домашним каталогом;
- для Windows два раза щелкните на файле с расширением `.exe` после того, как он скачается;
- для Linux выберите 32- или 64-битную версию. После скачивания запустите его (это большой сценарий оболочки).

Убедитесь, что имя скачиваемого вами файла начинается на Anaconda3. Если оно начинается с Anaconda, то это версия для Python 2.

Anaconda устанавливает содержимое в собственный каталог (он называется `anaconda` и располагается в домашнем каталоге). Это значит, что он не будет мешать другим версиям Python, установленным на ваш компьютер. Кроме того, вам не нужно особое разрешение (иметь имя, такое как `admin` или `root`), чтобы его установить.

Сейчас Anaconda содержит более 1500 пакетов с открытым исходным кодом. Посетите страницу документации установщика (https://docs.anaconda.com/anaconda/packages/pkg-docs) и нажмите ссылку для вашей платформы и версии Python.

После установки Anaconda 3 вы можете увидеть, что Санта положил в ваш компьютер, введя команду `conda list`.

Менеджер пакетов conda. Недавно разработчики Anaconda создали `conda` (https://docs.conda.io/), чтобы решить проблемы, которые возникали с `pip` и другими инструментами. `pip` — это менеджер пакетов для Python, а `conda` работает с любыми языками программирования и ПО. Этому пакету также не нужно ничего типа `virtualenv`, чтобы содержать отдельно разные пакеты.

Если вы уже установили дистрибутив Anaconda, то у вас уже есть программа `conda`. В противном случае можете установить Python 3 и `conda` со страницы miniconda (https://docs.conda.io/en/latest/miniconda.html). Как и в случае с Anaconda, убедитесь, что файл, который вы скачиваете, начинается с Miniconda3, а не с Miniconda (это версия для Python 2).

Пакет `conda` работает вместе с `pip`. Несмотря на то что он имеет собственный публичный репозиторий пакетов (https://anaconda.org/anaconda/repo), такие команды, как `conda search`, также выполняют поиск в репозитории PyPi (http://pypi.python.org/). Если у вас возникают проблемы с `pip`, то `conda` может стать хорошей альтернативой.

ПРИЛОЖЕНИЕ В

Нечто совершенно иное: async

Первые два приложения предназначались для начинающих программистов, а это пригодится тем, кто уже несколько продвинулся в данной области.

Как и большинство других языков программирования, Python был *синхронным*. Его код запускался последовательно по одной строке за раз, сверху вниз. Когда вы вызываете функцию, Python перескакивает в ее код, и вызывающая сторона ждет, пока функция не закончит выполняться, прежде чем возобновить то, что делалось ранее.

Ваш ЦП может совершать только одну операцию за раз, поэтому синхронное выполнение целесообразно. Но оказывается, зачастую программы не выполняют свой код, а ждут чего-то наподобие получения данных из файла или отклика сетевого сервиса. Процесс похож на то, как мы открываем окно браузера и ждем загрузки сайта. Если бы в наших силах было избежать этого «активного ожидания», то мы могли бы сократить общее время выполнения наших программ. Это также называется улучшением *пропускной способности*.

В главе 15 вы видели: при желании воспользоваться конкурентностью вам следует использовать процессы, потоки или сторонние решения, такие как `gevent` или `twisted`. Но сейчас появляется все больше *асинхронных* ответов, созданных на основе Python и сторонних решений. Они сосуществуют вместе с синхронным кодом Python, но, как предупреждали охотники за привидениями, вы не можете перекрещивать потоки. Я покажу, как можно избежать эктоплазматических сторонних эффектов.

Сопрограммы и циклы событий

В версию Python 3.4 был добавлен *асинхронный* модуль `asyncio`. Затем в версию Python 3.5 были добавлены ключевые слова `async` и `await`. Все это позволяет реализовать следующие новые концепции:

- *сопрограммы* — это функции, которые приостанавливаются в разных точках;
- *цикл событий* — цикл, в котором планируется запуск сопрограмм, а также они выполняются.

Это позволяет писать асинхронный код, выглядящий как обычный, синхронный, к которому мы привыкли. В противном случае следует использовать один из методов, описанных в главах 15 и 17, они подытожены ниже, в разделе «async против...» данного приложения.

Многозадачность — это то, что ваша операционная система делает с вашими процессами. Она решает, какое действие справедливо, кто занимает процессор, когда выполнять ввод/вывод данных и т. д. Цикл событий, однако, предоставляет возможность кооперативной многозадачности, при которой сопрограммы указывают, когда готовы стартовать и останавливаться. Они работают в одном потоке, поэтому вы не столкнетесь с потенциальными проблемами, описанными в подразделе «Потоки» на с. 311.

Сопрограмма *определяется* с помощью ключевого слова `async`, которое помещается перед `def`. Она *вызывается* следующим образом:

- путем размещения ключевого слова `await` перед ее именем в коде, что поместит сопрограмму в существующий цикл обработки событий. Вы можете сделать это только внутри другой сопрограммы;
- можно также использовать конструкцию `asyncio.run()`, которая явно запускает цикл обработки событий;
- помимо этого, можно воспользоваться вызовами `asyncio.create_task()` или `asyncio.ensure_future()`.

В этом примере показаны первые два метода вызова сопрограмм:

```
>>> import asyncio
>>>
>>> async def wicked():
...     print("Surrender,")
...     await asyncio.sleep(2)
...     print("Dorothy!")
...
>>> asyncio.run(wicked())
Surrender,
Dorothy!
```

В процессе выполнения программы произошла драматичная пауза длиной 2 секунды, которую вы не можете увидеть на печатной странице. Докажу, что я не жульничал (подробности работы модуля `timeit` см. в главе 19):

```
>>> from timeit import timeit
>>> timeit("asyncio.run(wicked())", globals=globals(), number=1)
Surrender,
Dorothy!
2.005701574998966
```

Вызов `asyncio.sleep(2)` сам по себе — сопрограмма, которую мы использовали в качестве примера некой времязатратной операции, такой как вызов API.

Строка `asyncio.run(wicked())` — это способ запуска сопрограммы из синхронного кода Python (на верхнем уровне программы).

Отличие от стандартного синхронного вызова (с использованием `time.sleep()`) заключается в том, что вызывающая сторона метода `wicked()` не блокируется на 2 секунды, пока работает метод.

Третий способ запустить сопрограмму — создать *задачу* и использовать ключевое слово `await`. Этот подход показан в следующем примере наряду с двумя предыдущими методами:

```
>>> import asyncio
>>>
>>> async def say(phrase, seconds):
...     print(phrase)
...     await asyncio.sleep(seconds)
...
>>> async def wicked():
...     task_1 = asyncio.create_task(say("Surrender,", 2))
...     task_2 = asyncio.create_task(say("Dorothy!", 0))
...     await task_1
...     await task_2
...
>>> asyncio.run(wicked())
Surrender,
Dorothy!
```

Запустив данный код, вы увидите: между выводом на экран двух строк нет задержки. Это произошло потому, что у нас есть две отдельные задачи. Первая — `task_1` — приостановилась на 2 секунды после вывода слова `Surrender`, но это не повлияло на вторую — `task_2`.

Ключевое слово `await` аналогично ключевому слову `yield` в генераторах, но вместо того, чтобы возвращать значение, оно помечает место, где цикл обработки событий может приостановиться, если нужно.

В документации (https://oreil.ly/Cf_hd) содержится гораздо больше информации. Синхронный и асинхронный код могут сосуществовать в одной программе. Главное — помнить о необходимости указывать ключевое слово `async` перед `def` при объявлении функции и ключевое слово `await` перед вызовом асинхронной функции.

Для дополнительной информации см.:

- список (https://oreil.ly/Vj0yD) ссылок на ПО, использующее `asyncio`;
- код (https://oreil.ly/n4FVx) веб-сканера, применяющего `asyncio`.

Альтернативы asyncio. Несмотря на то что модуль `asyncio` входит в стандартный пакет Python, вы можете использовать ключевые слова `async` и `await` без него. Сопрограммы и цикл обработки событий не зависят друг от друга. Дизайн модуля `asyncio` иногда подвергается критике (https://oreil.ly/n4FVx), и на этом фоне появились сторонние альтернативы:

- `curio` (https://curio.readthedocs.io/);
- `trio` (https://trio.readthedocs.io/).

Рассмотрим реальный пример, использующий `trio` и `asks` (https://asks.readthedocs.io/) (асинхронный веб-фреймворк, созданный на основе `requests` API). В при-

мере B.1 показывается веб-сканер, который использует конкурентность с помощью `trio` и `asks`, он был создан на основе ответа на stackoverflow (https://oreil.ly/CbINS). Чтобы запустить этот пример, сначала установите `trio` и `asks` с помощью `pip install`.

Пример B.1. trio_asks_sites.py

```
import time

import asks
import trio

asks.init("trio")

urls = [
    'https://boredomtherapy.com/bad-taxidermy/',
    'http://www.badtaxidermy.com/',
    'https://crappytaxidermy.com/',
    'https://www.ranker.com/list/bad-taxidermy-pictures/ashley-reign',
]

async def get_one(url, t1):
    r = await asks.get(url)
    t2 = time.time()
    print(f"{(t2-t1):.04}\t{len(r.content)}\t{url}")

async def get_sites(sites):
    t1 = time.time()
    async with trio.open_nursery() as nursery:
        for url in sites:
            nursery.start_soon(get_one, url, t1)

if __name__ == "__main__":
    print("seconds\tbytes\turl")
    trio.run(get_sites, urls)
```

Вот что я получил:

```
$ python trio_asks_sites.py
seconds bytes   url
0.1287  5735    https://boredomtherapy.com/bad-taxidermy/
0.2134  146082  https://www.ranker.com/list/bad-taxidermy-pictures/ashley-reign
0.215   11029   http://www.badtaxidermy.com/
0.3813  52385   https://crappytaxidermy.com/
```

Вы увидите, что `trio` вызывает не `asyncio.run()`, а собственный метод `trio.open_nursery()`. Если вам интересно, то можете прочесть эссе (https://oreil.ly/yp1-r) и обсуждение (https://oreil.ly/P21Ra) решений, связанных с дизайном `trio`.

Новый пакет, который называется AnyIO (https://anyio.readthedocs.io/en/latest), предоставляет единый интерфейс к `asyncio`, `curio` и `trio`.

В будущем можно ожидать появления новых асинхронных подходов как в стандартном Python, так и в сторонних библиотеках.

async против...

Как вы уже видели на протяжении этой книги, реализовать конкурентность можно с помощью множества приемов. Чем async отличается от них, а в чем похож?

- *Процессы*. Это хорошее решение, если вы хотите использовать все ядра ЦП на вашей машине или даже несколько машин. Однако процессы довольно тяжелые, на их запуск требуется некоторое время, а для коммуникации между процессами следует применить сериализацию.
- *Потоки*. Несмотря на то что потоки разрабатывались как «легковесная» альтернатива процессам, каждый поток занимает большой объем памяти. Сопрограммы гораздо легче, чем потоки; вы можете создать сотни тысяч сопрограмм на машине, которая может поддерживать всего несколько тысяч потоков.
- *«Зеленые» потоки*. Такие «зеленые» потоки, как `gevent`, работают хорошо и выглядят очень похоже на синхронный код, но требуют выполнять «обезьяний патч» стандартных функций Python, например библиотек сокетов.
- *Функции обратного вызова*. Такие библиотеки, как `twisted`, полагаются на функции обратного вызова: они вызываются, когда происходит определенное событие. Это знакомо программистам, работающим с GUI и JavaScript.
- *Очереди*. Как правило, являются крупномасштабным решением, когда вашим данным или процессам требуется больше одной машины.

Асинхронные фреймворки и серверы

Асинхронность появилась в Python недавно, и разработчикам требуется время для создания асинхронных версий таких фреймворков, как Flask.

Стандарт ASGI (https://asgi.readthedocs.io/) — это асинхронная версия WSGI, более подробно обсуждается здесь: https://oreil.ly/BnEXT.

Ниже представлены некоторые веб-серверы, работающие по принципу ASGI:

- `hypercorn` (https://pgjones.gitlab.io/hypercorn);
- `sanic` (https://sanic.readthedocs.io/);
- `uvicorn` (https://www.uvicorn.org/).

А вот некоторые асинхронные веб-фреймворки:

- `aiohttp` (https://aiohttp.readthedocs.io/) — клиент *и* сервер;
- `api_hour` (https://pythonhosted.org/api_hour);
- `asks` (https://asks.readthedocs.io/) — похож на `requests`;
- `blacksheep` (https://github.com/RobertoPrevato/BlackSheep);
- `bocadillo` (https://github.com/bocadilloproject/bocadillo);
- `channels` (https://channels.readthedocs.io/);
- `fastapi` (https://fastapi.tiangolo.com/) — использует аннотации типов;

- `muffin` (https://muffin.readthedocs.io/);
- `quart` (https://gitlab.com/pgjones/quart);
- `responder` (https://python-responder.org/);
- `sanic` (https://sanic.readthedocs.io/);
- `starlette` (https://www.starlette.io/);
- `tornado` (https://www.tornadoweb.org/);
- `vibora` (https://vibora.io/).

И наконец, несколько асинхронных интерфейсов к базам данных:

- `aiomysql` (https://aiomysql.readthedocs.io/);
- `aioredis` (https://aioredis.readthedocs.io/);
- `asyncpg` (https://github.com/magicstack/asyncpg).

ПРИЛОЖЕНИЕ Г
Ответы к упражнениям

1. Python: с чем его едят

1.1. Если вы еще не установили Python 3, сделайте это сейчас. Прочтите приложение Б, чтобы узнать детали.

1.2. Запустите интерактивный интерпретатор Python 3. Детали опять же вы найдете в приложении Б. 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 Mac:

```
$ python
Python 3.7.3 (v3.7.3:ef4ec6ed12, Mar 25 2019, 16:39:00)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

1.3. Немного поэкспериментируйте с интерпретатором. Используйте его как калькулятор и наберите `8 * 9`. Нажмите клавишу Enter, чтобы увидеть результат. Python должен вывести `72`.

```
>>> 8 * 9
72
```

1.4. Теперь введите число `47` и нажмите клавишу Enter. Появилось ли число `47` в следующей строке?

```
>>> 47
47
```

1.5. Теперь введите `print(47)` и нажмите клавишу Enter. Появилось ли снова число `47` в следующей строке?

```
>>> print(47)
47
```

2. Типы данных, значения, переменные и имена

2.1. Присвойте целочисленное значение `99` переменной `prince` и выведите ее на экран.

```
>>> prince = 99
>>> print(prince)
99
>>>
```

2.2. Какого типа значение `5`?

```
>>> type(5)
<class 'int'>
```

2.3. Какого типа значение `2.0`?

```
>>> type(2.0)
<class 'float'>
```

2.4. Какого типа выражение `5 + 2.0`?

```
>>> type(5 + 2.0)
<class 'float'>
```

3. Числа

3.1. Сколько секунд в часе? Используйте интерактивный интерпретатор как калькулятор и умножьте количество секунд в минуте (60) на количество минут в часе (тоже 60).

```
>>> 60 * 60
3600
```

3.2. Присвойте результат вычисления предыдущего задания (секунды в часе) переменной, которая называется `seconds_per_hour`.

```
>>> seconds_per_hour = 60 * 60
>>> seconds_per_hour
3600
```

3.3. Сколько секунд в сутках? Используйте переменную `seconds_per_hour`.

```
>>> seconds_per_hour * 24
86400
```

3.4. 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
```

3.5. Разделите значение переменной `seconds_per_day` на значение переменной `seconds_per_hour`, используя деление с плавающей точкой (`/`).

```
>>> seconds_per_day / seconds_per_hour
24.0
```

3.6. Разделите значение переменной `seconds_per_day` на значение переменной `seconds_per_hour`, используя целочисленное деление (`//`). Совпадает ли полученный результат с ответом из предыдущего пункта без учета символов `.0` в конце?

```
>>> seconds_per_day // seconds_per_hour
24
```

4. Выбираем с помощью if

4.1. Выберите число от 1 до 10 и присвойте его переменной `secret`. Выберите еще одно число от 1 до 10 и присвойте его переменной `guess`. Далее напишите условные проверки (`if`, `else` и `elif`), чтобы вывести строку `'too low'`, если значение переменной `guess` меньше `7`, `'too high'`, если оно больше `7`, и `'just right'`, если равно `secret`.

Выбрали ли вы для переменной `secret` значение `7`? Готов поспорить, так сделали многие, поскольку в числе `7` есть нечто волшебное.

```
secret = 7
guess = 5
if guess < secret:
    print('too low')
elif guess > secret:
    print('too high')
else:
    print('just right')
```

Запустите эту программу, и вы увидите следующее:

```
too low
```

4.2. Присвойте значения `True` или `False` переменным `small` и `green`. Напишите несколько утверждений `if/else`, которые выведут на экран информацию о том, соответствуют ли заданным условиям следующие растения: вишня, горошек, арбуз, тыква.

```
>>> small = False
>>> green = True
>>> if small:
...     if green:
...         print("pea")
...     else:
...         print("cherry")
... else:
...     if green:
...         print("watermelon")
...     else:
...         print("pumpkin")
...
Watermelon
```

5. Текстовые строки

5.1. Напишите с заглавной буквы слово, которое начинается с буквы m:

```
>>> song = """When an eel grabs your arm,
... And it causes great harm,
... That's - a moray!"""
```

Не забудьте о пробеле, стоящем перед буквой `m`:

```
>>> song = """When an eel grabs your arm,
... And it causes great harm,
... That's - a moray!"""
>>> song = song.replace(" m", " M")
>>> print(song)
When an eel grabs your arm,
And it causes great harm,
That's - a Moray!
```

5.2. Выведите на экран все вопросы из списка, а также правильные ответы в таком виде:

Q: *вопрос*

A: *ответ*

```
>>> questions = [
...     "We don't serve strings around here. Are you a string?",
...     "What is said on Father's Day in the forest?",
...     "What makes the sound 'Sis! Boom! Bah!'?"
...     ]
>>> answers = [
```

```
...     "An exploding sheep.",
...     "No, I'm a frayed knot.",
...     "'Pop!' goes the weasel."
...     ]
```

Вы могли бы вывести на экран все вопросы и ответы множеством способов. Попробуем использовать «бутерброд из кортежей» (конструкция, в которой в кортежах хранятся кортежи) с целью их сгруппировать, а затем распакуем кортежи, чтобы получить значения для вывода на экран:

```
questions = [
    "We don't serve strings around here. Are you a string?",
    "What is said on Father's Day in the forest?",
    "What makes the sound 'Sis! Boom! Bah!'?"
    ]
answers = [
    "An exploding sheep.",
    "No, I'm a frayed knot.",
    "'Pop!' goes the weasel."
    ]

q_a = ( (0, 1), (1,2), (2, 0) )
for q, a in q_a:
    print("Q:", questions[q])
    print("A:", answers[a])
    print()
```

Результат:

```
$ python qanda.py
Q: We don't serve strings around here. Are you a string?
A: No, I'm a frayed knot.

Q: What is said on Father's Day in the forest?
A: 'Pop!' goes the weasel.

Q: What makes the sound 'Sis! Boom! Bah!'?
A: An exploding sheep.
```

5.3. Выведите на экран следующее стихотворение, используя старый стиль форматирования. Подставьте в него такие строки: `'roast beef'`, `'ham'`, `'head'`, и `'clam'`:

```
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.
```

5.4. Напишите письмо с использованием нового стиля форматирования. Сохраните предложенную строку в переменной `letter` (она понадобится вам в упражнении ниже):

```
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}
... {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}
... '''
```

5.5. Присвойте значения переменным `'salutation'`, `'name'`, `'product'`, `'verbed'` (глагол в прошедшем времени), `'room'`, `'animals'`, `'percent'`, `'spokesman'` и `'job_title'`. С помощью функции `letter.format()` выведите на экран значение переменной `letter`, в которую подставлены эти значения.

```
>>> print (
...     letter.format(salutation='Ambassador',
...                   name='Nibbler',
...                   product='pudding',
...                   verbed='evaporated',
...                   room='gazebo',
...                   animals='octothorpes',
...                   amount='$1.99',
...                   percent=88,
...                   spokesman='Shirley Iugeste',
...                   job_title='I Hate This Job')
...     )

Dear Ambassador Nibbler,

Thank you for your letter. We are sorry that our pudding
evaporated in your gazebo. Please note that it should never
be used in a gazebo, especially near any octothorpes.

Send us your receipt and $1.99 for shipping and handling.
We will send you another pudding that, in our tests,
is 88% less likely to have evaporated.

Thank you for your support.

Sincerely,
Shirley Iugeste
I Hate This Job
```

5.6. После проведения публичных опросов с целью выбора имени появились: английская подводная лодка Boaty McBoatface, австралийская беговая лошадь Horsey McHorseface и шведский поезд Trainy McTrainface. Используйте форматирование с символом `%` для того, чтобы вывести на экран победившие имена для утки, тыквы и шпица (пример Д.1).

Пример Д.1. mcnames1.py

```
names = ["duck", "gourd", "spitz"]
for name in names:
    cap_name = name.capitalize()
    print("%sy Mc%sface" % (cap_name, cap_name))

Output:

    Ducky McDuckface
    Gourdy McGourdface
    Spitzy McSpitzface
```

5.7. Сделайте то же самое с помощью функции `format()` (пример Д.2).

Пример Д.2. mcnames2.py

```
names = ["duck", "gourd", "spitz"]
for name in names:
   cap_name = name.capitalize()
    print("{}y Mc{}face".format(cap_name, cap_name))
```

5.8. А теперь еще раз с использованием *f-строк* (пример Д.3).

Пример Д.3. mcnames3.py

```
names = ["duck", "gourd", "spitz"]
for name in names:
    cap_name = name.capitalize()
    print(f"{cap_name}y Mc{cap_name}face")
```

6. Создаем циклы с помощью ключевых слов while и for

6.1. 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
```

6.2. Присвойте значение `7` переменной `guess_me` и значение `1` переменной `number`. Напишите цикл `while`, который сравнивает переменные `number` и `guess_me`. Выведите строку `'too low'`, если значение переменной `number` меньше значения переменной `guess_me`. Если значение переменной `number` равно значению переменной `guess_me`, выведите строку `'found it!'` и выйдите из цикла. Если значение переменной `number` больше значения переменной `guess_me`, выведите строку `'oops'` и выйдите из цикла. Увеличьте значение переменной `number` на выходе из цикла.

```
guess_me = 7
number = 1
while True:
    if number < guess_me:
        print('too low')
    elif number == guess_me:
        print('found it!')
       break
    elif number > guess_me:
        print('oops')
        break
    number += 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: строка `elif number > guess_me:` могла содержать обычный оператор `else:` в связи с тем, что если значение `number` не меньше и не равно значению `guess_me`, то оно должно быть больше. По крайней мере в этой Вселенной.

6.3. Присвойте значение `5` переменной `guess_me`. Используйте цикл `for`, чтобы проитерировать с помощью переменной `number` по диапазону `range(10)`. Если значение переменной `number` меньше, чем значение `guess_me`, выведите на экран сообщение `'too low'`. Если оно равно значению `guess_me` — выведите сообщение `found it!`, а затем выйдите из цикла. Если значение переменной `number` больше, чем `guess_me`, выведите на экран сообщение `'oops'` и выйдите из цикла.

```
>>> guess_me = 5
>>> for number in range(10):
...     if number < guess_me:
...         print("too low")
...     elif number == guess_me:
...         print("found it!")
...         break
...     else:
...         print("oops")
...         break
...
too low
too low
too low
too low
too low
found it!
```

7. Кортежи и списки

7.1. 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, то список будет выглядеть так: `years_list = [1980, 1981, 1982, 1983, 1984, 1985]`.

Если вы родились в 1980-м, то вам нужно ввести следующее:

```
>>> years_list = [1980, 1981, 1982, 1983, 1984, 1985]
```

7.2. В какой год из списка `years_list` был ваш третий день рождения? Учтите, в первый год вам было 0 лет.

Вам нужно смещение `3`. Поэтому, если вы родились в 1980-м:

```
>>> years_list[3]
1983
```

7.3. В какой год из списка `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
```

7.4. Создайте список `things`, содержащий три элемента: `"mozzarella"`, `"cinderella"`, `"salmonella"`.

```
>>> things = ["mozzarella", "cinderella", "salmonella"]
>>> things
['mozzarella', 'cinderella', 'salmonella']
```

7.5. Напишите с большой буквы тот элемент списка `things`, который означает человека, 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']
```

7.6. Переведите сырный элемент списка `things` в верхний регистр целиком и выведите список.

```
>>> things[0] = things[0].upper()
>>> things
['MOZZARELLA', 'Cinderella', 'salmonella']
```

7.7. Удалите из списка `things` заболевание, 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]
```

7.8. Создайте список с элементами `"Groucho"`, `"Chico"` и `"Harpo"`; назовите его `surprise`.

```
>>> surprise = ['Groucho', 'Chico', 'Harpo']
>>> surprise
['Groucho', 'Chico', 'Harpo']
```

7.9. Напишите последний элемент списка `surprise` со строчной буквы, затем выведите эту строку в обратном порядке и с прописной буквы.

```
>>> surprise[-1] = surprise[-1].lower()
>>> surprise[-1] = surprise[-1][::-1]
>>> surprise[-1].capitalize()
'Oprah'
```

7.10. Используйте списковое включение, чтобы создать список с именем `even`, в котором будут содержаться четные числа в промежутке `range(10)`.

```
>>> even = [number for number in range(10) if number % 2 == 0]
>>> even
[0, 2, 4, 6, 8]
```

7.11. Попробуйте создать генератор рифм для прыжков через скакалку. Нужно вывести на экран последовательность двухстрочных рифм. Начните программу с этого фрагмента:

```
start1 = ["fee", "fie", "foe"]
rhymes = [
    ("flop", "get a mop"),
    ("fope", "turn the rope"),
    ("fa", "get your ma"),
```

```
    ("fudge", "call the judge"),
    ("fat", "pet the cat"),
    ("fog", "walk the dog"),
    ("fun", "say we're done"),
    ]
start2 = "Someone better"
```

Затем следуйте инструкциям.

Для каждого кортежа (`first, second`) в списке `rhymes`:

- для первой строки:
  - выведите на экран каждую строку списка `start1`, записав их с большой буквы, за которыми следует восклицательный знак и пробел;
  - выведите на экран значение переменной `first`, также записав его с большой буквы, за которым следует восклицательный знак;
- для второй строки:
  - выведите на экран значение переменной `start2` и пробел;
  - выведите на экран значение переменной `second` и точку.

```
start1 = ["fee", "fie", "foe"]
rhymes = [
    ("flop", "get a mop"),
    ("fope", "turn the rope"),
    ("fa", "get your ma"),
    ("fudge", "call the judge"),
    ("fat", "pet the cat"),
    ("fog", "pet the dog"),
    ("fun", "say we're done"),
    ]
start2 = "Someone better"
start1_caps = " ".join([word.capitalize() + "!" for word in start1])
for first, second in rhymes:
    print(f"{start1_caps} {first.capitalize()}!")
    print(f"{start2} {second}.")
```

Результат:

```
Fee! Fie! Foe! Flop!
Someone better get a mop.
Fee! Fie! Foe! Fope!
Someone better turn the rope.
Fee! Fie! Foe! Fa!
Someone better get your ma.
Fee! Fie! Foe! Fudge!
Someone better call the judge.
Fee! Fie! Foe! Fat!
Someone better pet the cat.
Fee! Fie! Foe! Fog!
Someone better walk the dog.
Fee! Fie! Foe! Fun!
Someone better say we're done.
```

8. Словари и множества

8.1. Создайте англо-французский словарь с названием `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'}
```

8.2. Используя словарь `e2f`, выведите французский вариант слова `walrus`.

```
>>> e2f['walrus']
'morse'
```

8.3. 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'}
```

8.4. Используя словарь `f2e`, выведите английский вариант слова `chien`.

```
>>> f2e['chien']
'dog'
```

8.5. Выведите на экран множество английских слов из ключей словаря `e2f`.

```
>>> set(e2f.keys())
{'cat', 'walrus', 'dog'}
```

8.6. 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, так что если вы заглянули сюда, то ничего особо страшного не случилось:

```
>>> life = {
...     'animals': {
...         'cats': [
...             'Henri', 'Grumpy', 'Lucy'
...             ],
```

```
...         'octopi': {},
...         'emus': {}
...         },
...     'plants': {},
...     'other': {}
...     }
>>>
```

8.7. 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 читабельным:

```
>>> print ( list ( life.keys() ) )
['animals', 'other', 'plants']
```

8.8. Выведите на экран ключи `life['animals']`.

```
>>> print(life['animals'].keys())
dict_keys(['cats', 'octopi', 'emus'])
```

8.9. Выведите значения `life['animals']['cats']`.

```
>>> print(life['animals']['cats'])
['Henri', 'Grumpy', 'Lucy']
```

8.10. Используйте генератор словаря, чтобы создать словарь `squares`. Используйте `range(10)`, чтобы получить ключи. В качестве значений используйте возведенное в квадрат значение каждого ключа.

```
>>> 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}
```

8.11. Используйте генератор множества, чтобы создать множество `odd` из нечетных чисел диапазона `range(10)`.

```
>>> odd = {number for number in range(10) if number % 2 == 1}
>>> odd
{1, 3, 5, 7, 9}
```

8.12. Используйте включение генератора, чтобы вернуть строку `'Got'` и числа из диапазона `range(10)`. Итерируйте по этой конструкции 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
```

8.13. Используйте функцию `zip()`, чтобы создать словарь из кортежа ключей (`'optimist'`, `'pessimist'`, `'troll'`) и кортежа значений (`'The glass is half full'`, `'The glass is half empty'`, `'How did you get a glass?'`).

```
>>> keys = ('optimist', 'pessimist', 'troll')
>>> values = ('The glass is half full',
...     'The glass is half empty',
...     'How did you get a glass?')
>>> dict(zip(keys, values))
{'optimist': 'The glass is half full',
'pessimist': 'The glass is half empty',
'troll': 'How did you get a glass?'}
```

8.14. Используйте функцию `zip()`, чтобы создать словарь с именем `movies`, в котором объединены списки `titles = ['Creature of Habit', 'Crewel Fate', 'Sharks On a Plane']` и `plots = ['A nun turns into a monster', 'A haunted yarn shop', 'Check your exits']`.

```
>>> titles = ['Creature of Habit',
...     'Crewel Fate',
...     'Sharks On a Plane']
>>> plots = ['A nun turns into a monster',
...     'A haunted yarn shop',
...     'Check your exits']
>>> movies = dict(zip(titles, plots))
>>> movies
{'Creature of Habit': 'A nun turns into a monster',
'Crewel Fate': 'A haunted yarn shop',
'Sharks On a Plane': 'Check your exits'}
>>>
```

9. Функции

9.1. Определите функцию `good`, которая возвращает следующий список: `['Harry', 'Ron', 'Hermione']`.

```
>>> def good():
...     return ['Harry', 'Ron', 'Hermione']
...
>>> good()
['Harry', 'Ron', 'Hermione']
```

9.2. 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
...
>>> count = 1
>>> for number in get_odds():
...     if count == 3:
...         print("The third odd number is", number)
...         break
...     count += 1
...
```

Третье нечетное число равно `5`.

9.3. Определите декоратор `test`, который выводит строку `'start'` при вызове функции 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
```

9.4. Определите исключение `OopsException`. Сгенерируйте его и посмотрите, 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
```

10. Ой-ой-ой: объекты и классы

10.1. Создайте класс `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>
```

10.2. Создайте новый класс с именем `Thing2` и присвойте переменной `letters` значение `'abc'`. Выведите на экран значение `letters`.

```
>>> class Thing2:
...     letters = 'abc'
...
>>> print(Thing2.letters)
abc
```

10.3. Создайте еще один класс, который, конечно же, называется `Thing3`. В этот раз присвойте значение `'xyz'` переменной объекта `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
```

10.4. Создайте класс `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)
```

10.5. 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 имени совпадает с аргументами функции `__init__` (аргументы — ключевые слова рассматриваются в главе 9):

```
>>> hydrogen = Element(**el_dict)
>>> hydrogen.name
'Hydrogen'
```

10.6. 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
```

10.7. 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
```

Метод `__str__()` — это один из волшебных методов Python. Функция `print` вызывает метод объекта `__str__()`, чтобы получить его строковое представление. Если у объекта нет метода `__str__()`, то он получает метод по умолчанию от его родительского класса `Object`, который возвращает строку наподобие `<__main__.Element object at 0x1006f5310>`.

10.8. Модифицируйте класс `Element`, сделав атрибуты `name`, `symbol` и `number` приватными. Определите свойство получателя для каждого атрибута, возвращающее его значение.

```
>>> 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
```

10.9. 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 ест указанное животное.

```
>> 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
```

10.10. 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`,

содержащий 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.
```

11. Модули, пакеты и программы

11.1. Создайте файл с именем `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()`.

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
```

11.2. В интерактивном интерпретаторе импортируйте модуль `zoo` под именем `menagerie` и вызовите его функцию `hours()`.

```
>>> import zoo as menagerie
>>> menagerie.hours()
Open 9-5 daily
```

11.3. Оставаясь в интерпретаторе, импортируйте непосредственно функцию `hours()` из модуля `zoo` и вызовите ее.

```
>>> from zoo import hours
>>> hours()
Open 9-5 daily
```

11.4. Импортируйте функцию `hours()` под именем `info` и вызовите ее.

```
>>> from zoo import hours as info
>>> info()
Open 9-5 daily
```

11.5. 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}
```

11.6. Создайте `OrderedDict` с именем `fancy` из пар «ключ — значение», приведенных в упражнении 11.5, 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)])
```

11.7. 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']
```

12. Обрабатываем данные

12.1. Создайте строку Unicode с именем `mystery` и присвойте ей значение `'\U0001f4a9'`. Выведите на экран значение `mystery` и ее имя Unicode.

```
>>> import unicodedata
>>> mystery = '\U0001f984'
>>> mystery
'💩'
>>> unicodedata.name(mystery)
'PILE OF POO'
```

Ой-ой-ой! Что еще у них там есть?

12.2. Закодируйте строку `mystery`, на 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'
```

12.3. Используя кодировку UTF-8, декодируйте переменную `popbytes` 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
```

12.4. 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 в 1866 году во славу головки сыра весом 7000 фунтов, которая была изготовлена в Онтарио и отправлена в международное путешествие. Если не хотите самостоятельно перепечатывать это стихотворение, используйте свой любимый поисковик и скопируйте текст в программу. Или скопируйте стихотворение из проекта «Гутенберг» (http://bit.ly/mcintyre-poetry). 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.
... '''
```

12.5. Импортируйте модуль `re`, чтобы использовать функции регулярных выражений в Python. Примените функцию `re.findall()` для вывода на экран всех слов, которые начинаются с буквы `c`.

Мы определим переменную `pat` для шаблона и затем будем искать такой шаблон в строке `mammoth`:

```
>>> import re
>>> pat = r'\bc\w*'
>>> re.findall(pat, mammoth)
['cheese', 'city', 'cheese', 'cheek', 'could', 'cheese', 'cast', 'crush']
```

Элемент `\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. Элемент `\w` означает любой символ слова, которое включает в себя буквы, цифры и подчеркивания (_). Элемент `*` означает ноль или больше таких символов. Целиком это выражение находит слова, которые начинаются с `c`, включая `'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)
[]
```

12.6. 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']
```

12.7. 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 на осознание: 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']
```

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 на осознание: 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']
```

12.8. Найдите все слова, которые содержат три гласные подряд.

Начиная с границы слова, любое количество символов *слова*, три гласные и далее любые символы, не являющиеся гласными, до конца слова:

```
>>> pat = r'\b[^aeiou]*[aeiou]{3}[^aeiou]*\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']
```

На сей 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 то, что регулярные выражения могут пригодиться для многого, но их может быть очень трудно написать правильно.

12.9. Используйте метод `unhexlify` для того, чтобы преобразовать эту шестнадцатеричную строку, созданную путем объединения двух строк для размещения на странице, в переменную типа `bytes` с именем `gif`.

```
'4749463839610100010080000000000ffffff21f9' +
'0401000000002c000000000100010000020144003b'
```

```
>>> import binascii
>>> hex_str = '4749463839610100010080000000000ffffff21f9' + \
...     '0401000000002c000000000100010000020144003b'
>>> gif = binascii.unhexlify(hex_str)
>>> len(gif)
42
```

12.10. 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`. Корректен ли этот файл?

```
>>> gif[:6] == b'GIF89a'
True
```

Обратите внимание: с помощью `b` мы можем указать, что строка состоит из байтов, а не из символов Unicode. Вы можете сравнить байты с байтами, но не байты и символы:

```
>>> gif[:6] == 'GIF89a'
False
>>> type(gif)
<class 'bytes'>
>>> type('GIF89a')
<class 'str'>
>>> type(b'GIF89a')
<class 'bytes'>
```

12.11. Ширина GIF-файла в пикселах является шестнадцатибитным целым числом с обратным порядком байтов, которое начинается со смещения 6 байт. 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)
```

13. Календари и часы

13.1. 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.txt', 'wt') as output:
...    print(now_str, file=output)
>>>
```

Когда я запустил этот фрагмент кода, в файле `today.txt` увидел следующее:

```
2019-07-23
```

Вместо функции `print` вы могли бы задействовать такую строку, как `output.write(now_str)`. Использование функции `print` добавляет символ перевода строки в конце.

13.2. Прочтите текстовый файл `today.txt` и разместите данные в строке `today_string`.

```
>>> with open('today.txt', 'rt') as input:
...     today_string = input.read()
...
>>> today_string
'2019-07-23\n'
```

13.3. Проанализируйте дату из строки `today_string`.

```
>>> fmt = '%Y-%m-%d\n'
>>> datetime.strptime(today_string, fmt)
datetime.datetime(2019, 7, 23, 0, 0)
```

Если вы записали тот символ новой строки в файл, то вам нужно, чтобы он совпал со строкой формата.

13.4. Создайте объект `date` с датой вашего рождения.

Предположим, вы родились 14 августа 1982 года:

```
>>> my_day = date(1982, 8, 14)
>>> my_day
datetime.date(1982, 8, 14)
```

13.5. 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.

13.6. 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, то вы, возможно, пропустили еще один повод повеселиться.

14. Файлы и каталоги

14.1. Выведите на экран список файлов текущего каталога.

Если ваш текущий каталог называется `ohmy` и содержит три файла с именами по названиям животных, то код может выглядеть так:

```
>>> import os
>>> os.listdir('.')
['bears', 'lions', 'tigers']
```

14.2. Выведите на экран список всех файлов родительского каталога.

Если родительский каталог содержит два файла и текущий каталог `ohmy`, то код может выглядеть так:

```
>>> import os
>>> os.listdir('..')
['ohmy', 'paws', 'whiskers']
```

14.3. Присвойте строку `This is a test of the emergency text system` переменной `test1` и запишите эту переменную в файл с именем `test.txt`.

```
>>> test1 = 'This is a test of the emergency text system'
>>> len(test1)
43
```

Вот как это можно сделать 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
```

14.4. Откройте файл `test.txt` и считайте его содержимое в строку `test2`. Будут ли одинаковыми строки `test1` и `test2`?

```
>>> with open('test.txt', 'rt') as infile:
...     test2 = infile.read()
...
>>> len(test2)
43
>>> test1 == test2
True
```

15. Данные во времени: процессы и конкурентность

15.1. Используйте модуль `multiprocessing`, чтобы создать три отдельных процесса. Заставьте каждый из них подождать случайное количество секунд между нулем и единицей, вывести текущее время, а затем завершить работу.

```
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 0.10720361113059229 seconds, time is 2019-07-24 00:19:23.951594
wait 0.5825144002370065 seconds, time is 2019-07-24 00:19:24.425047
wait 0.6647690569029477 seconds, time is 2019-07-24 00:19:24.509995
```

16. Данные в коробке: устойчивые хранилища

16.1. Сохраните следующие несколько строк в файл `books.csv`. Обратите внимание на то, что, если поля разделены запятыми, вам нужно заключить в кавычки поле, содержащее 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
```

16.2. 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('books.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'}
```

16.3. 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
```

```
>>> 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('books2.csv', 'wt') as outfile:
...     outfile.write(text)
...
201
```

16.4. Используйте модуль `sqlite3`, чтобы создать базу данных SQLite `books.db` и таблицу `books`, содержащую следующие поля: `title` (текст), `author` (текст) и `year` (целое значение).

```
>>> 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()
```

16.5. Считайте данные из файла `books2.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()
```

16.6. Считайте и выведите на экран столбец `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), то 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, то 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
```

16.7. Считайте и выведите на экран все столбцы 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 столбцы каждой строки, 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
```

16.8. Используйте модуль `sqlalchemy`, чтобы подключиться к базе данных `sqlite3 books.db`, которую вы создали в упражнении 16.4. Как и в упражнении 16.6, считайте и выведите на экран столбец `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!',)
```

16.9. 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'}
```

16.10. Увеличьте поле `count` хеша `test` и выведите его на экран.

```
>>> conn.hincrby('test', 'count', 3)
4
>>> conn.hget('test', 'count')
b'4'
```

17. Данные в пространстве: сети

17.1. Используйте объект класса `socket`, чтобы реализовать сервис, сообщающий 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 программы работают вечно, поэтому вам нужно завершать их вручную.

17.2. Задействуйте сокеты ZeroMQ REQ и REP, чтобы сделать то же самое.

```
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:
    #  ожидаем следующего запроса от клиента
    message = server.recv()
    if message == b'time':
        now = datetime.utcnow()
        reply = str(now)
        server.send(bytes(reply, 'utf-8'))
        print('Server sent', reply)

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'
```

17.3. Попробуйте сделать то же самое с помощью XMLRPC.

С сервера:

```
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()
```

С клиента:

```
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
```

17.4. Возможно, вы видели эпизод телесериала *I Love Lucy*, в котором Люси и Этель работают на шоколадной фабрике. Парочка стала отставать, когда линия конвейера, направлявшая к ним на обработку конфеты, еще более ускорилась. Напишите симуляцию, которая отправляет разные типы конфет в список Redis, и клиент Lucy, делающий 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:

Симуляция `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)
```

Клиент `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 0,5 секунды для обработки каждой конфеты и они появляются в среднем каждые 0,5 секунды, 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.

17.5. Используйте ZeroMQ, чтобы публиковать стихотворение из упражнения 12.4 (пример 12.1) по одному слову за раз. Напишите потребитель ZeroMQ, который будет выводить на экран каждое слово, начинающееся с гласной. Напишите другой потребитель, который станет 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. Одни слова могут оказаться в обеих темах, другие — 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, то они *не будут* работать, хотя код выглядит хорошо. Вам нужно прочитать руководство ZeroMQ (http://zguide.zeromq.org/page:all), чтобы узнать о проблеме *медленного присоединившегося*: даже если вы запустите клиент раньше сервера, тот 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, то это не проблема. Но в таком случае поток данных настолько короткий, что заканчивается еще до того, как подписчик успеет моргнуть.

Простейший способ исправить это — заставить публикатор пропустить секунду после вызова метода `bind()` и до того, как он начнет отправлять сообщения. Назовем данную 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 подписчик, а затем и «сонный» публикатор:

```
$ 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, то синхронизируйте публикатор и подписчик с помощью сокетов `REQ` и `REP`. Примеры файлов `publisher.py` и `subscriber.py` вы можете найти на GitHub (http://bit.ly/pyzmq-gh).

18. Распутываем Всемирную паутину

18.1. Если вы еще не установили Flask, то сделайте это сейчас. Это также позволит установить `werkzeug`, `jinja2` и, возможно, другие пакеты.

18.2. 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 этот порт для чего-то еще, то 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
```

18.3. Добавьте функцию `home()` для обработки запросов к главной странице. Пусть она возвращает строку `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 главной странице через браузер, HTTP-программы командной строки, такие как `curl`, или `wget`, или даже `telnet`:

```
$ curl http://localhost:5000/
It's alive!
```

18.4. 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>
```

Создайте каталог `templates` и файл `home.html`, содержащий показанное. Если ваш сервер Flask все еще работает после запуска предыдущих примеров, то обнаружит новый контент и перезапустится.

18.5. 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 вот что:

```
I'm of course referring to Octothorpe, which is 7 feet tall and green.
```

19. Быть питонщиком

Питонщикам сегодня ничего не задавали.

20. Пи-Арт

20.1. Установите `matplotlib`. Нарисуйте диаграмму рассеивания для следующих пар `(x, y)`: `( (0, 0), (3,5), (6, 2), (9, 8), (14, 10) )`.

20.2. Нарисуйте линейчатый график на основе тех же данных.

20.3. Нарисуйте график на основе тех же данных.

В этом фрагменте кода показаны все три подграфика:

```
import matplotlib.pyplot as plt

x = (0, 3, 6, 9, 14)
y = (0, 5, 2, 8, 10)
fig, plots = plt.subplots(nrows=1, ncols=3)

plots[0].scatter(x, y)
plots[1].plot(x, y)
plots[2].plot(x, y, 'o-')

plt.show()
```

21. За работой

21.1. Установите Geopandas и запустите пример 21.1. Попробуйте поизменять цвета и размеры меток.

22. Python в науке

22.1. Установите Pandas. Получите CSV-файл в примере 16.1. Запустите программу из примера 16.2. Поэкспериментируйте с командами Pandas.

ПРИЛОЖЕНИЕ Д
Вспомогательные таблицы

Я обнаружил, что кое-какие вещи мне приходится подсматривать слишком часто. Вот несколько таблиц, которые, надеюсь, окажутся вам полезны.

Приоритет операторов

Таблица Д.1 — ремикс официальной документации о приоритетах для Python 3, операторы с самым высоким приоритетом находятся наверху.

Таблица Д.1. Приоритет операторов

Оператор	**Описание и примеры**
[v, ...], {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
+n, –n, ~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* (можно применять с любым объектом `str`) и модуль `string`, содержащий полезные определения. Задействуем проверочные переменные:

```
>>> s = "OH, my paws and whiskers!"
>>> t = "I'm late!"
```

В следующих примерах оболочка Python выведет результат вызова метода, но оригинальные переменные `s` и `t` не изменятся.

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(' ', 1)
['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 (табл. Д.2).

Таблица Д.2. Атрибуты для определения констант

Атрибут	Пример
ascii_letters	'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'
ascii_lowercase	'abcdefghijklmnopqrstuvwxyz'
ascii_uppercase	'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
digits	'0123456789'
hexdigits	'0123456789abcdefABCDEF'
octdigits	'01234567'
punctuation	'!"#$%&\'()*+,-./:;<=>?@[\\]^_\{|}~'`
printable	цифры + ascii_символы + пунктуация + пробел
whitespace	' \t\n\r\x0b\x0c'

Эпилог

Честер хотел бы выразить вам уважение за ваше усердие. Если он вам нужен, то знайте, что сейчас он спит...

Рис. Д.1. Честер[1]

...но Люси готова ответить на любые вопросы.

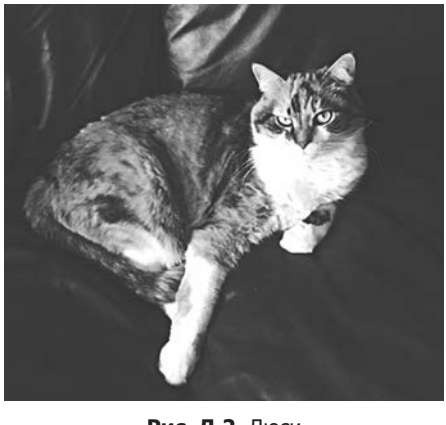

Рис. Д.2. Люси

[1] Он сместился примерно на фут вправо по сравнению с рис. 3.1.

Билл Любанович

Простой Python. Современный стиль программирования

2-е издание

Перевел с английского *Е. Зазноба*

Заведующая редакцией	*Ю. Сергиенко*
Ведущий редактор	*Н. Гринчик*
Литературные редакторы	*Н. Кудрейко, Н. Хлебина*
Художественный редактор	*В. Мостипан*
Корректоры	*Е. Павлович, Е. Рафалюк-Бузовская*
Верстка	*Г. Блинов*

Изготовлено в России. Изготовитель: ООО «Прогресс книга».
Место нахождения и фактический адрес: 194044, Россия, г. Санкт-Петербург,
Б. Сампсониевский пр., д. 29А, пом. 52. Тел.: +78127037373.

Дата изготовления: 10.2020. Наименование: книжная продукция. Срок годности: не ограничен.

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

Импортер в Беларусь: ООО «ПИТЕР М», 220020, РБ, г. Минск, ул. Тимирязева, д. 121/3, к. 214, тел./факс: 208 80 01.

Подписано в печать 14.10.20. Формат 70×100/16. Бумага офсетная. Усл. п. л. 47,730. Тираж 1000. Заказ 0000.